SO2pin, a C++ library for Yukawa decomposition in SO($2N$) models

Nuno Cardoso[a,b], David Emmanuel-Costa[b,*], Nuno Gonçalves[c], C. Simões[d]

[a] *NCSA, University of Illinois at Urbana-Champaign, 1205 W. Clark St., Urbana, IL 61801, USA*
[b] *CFTP, Departamento de Física, Instituto Superior Técnico, Universidade de Lisboa, Av. Rovisco Pais, 1049-001 Lisbon, Portugal*
[c] *Institute of Systems and Robotics, Department of Electrical and Computer Engineering, University of Coimbra, Polo 2 - Pinhal de Marrocos, 3030-290 Coimbra, Portugal*
[d] *IFPA, Dép. AGO, Quartier Agora, 19A Allée du 6 août, Bât B5a, Université de Liège, 4000 Liège, Belgique*

Abstract

We present in this paper the SO2pin library, which calculates an analytic decomposition of the Yukawa interactions invariant under SO($2N$) in terms of an SU(N) basis. We make use of the oscillator expansion formalism, where the SO($2N$) spinor representations are expressed in terms of creation and annihilation operators of a Grassmann algebra acting on a vacuum state. These noncommutative operators and their products are simulated in SO2pin through the implementation of doubly-linked-list data structures. These data structures were determinant to achieve a higher performance in the simplification of large products of creation and annihilation operators. We illustrate the use of our library with complete examples of how to decompose Yukawa terms invariant under SO($2N$) in terms of SU(N) degrees of freedom for $N = 2$ and 5. We further demonstrate, with an example for SO(4), that higher dimensional field-operator terms can also be processed with our library. Finally, we describe the functions available in SO2pin that are made to simplify the writing of spinors and their interactions specifically for SO(10) models.

Keywords: Special orthogonal groups, Grand Unified Theory
PACS: 02.20.Qs, 02.70.Wz, 12.10.-g, 12.10.Dm
Prepint: CFTP/15-008
Version: 1.0

Contents

1. Introduction

The orthogonal groups O(N) and their generalisations have played an important role in the construction of modern physics. In particular, the special orthogonal groups SO(N) appear naturally in the context of physical systems invariant under rotations, which in turn implies the conservation of the angular momentum or the determination of the azimuthal quantum number for an atomic orbital. The notion of spin used to describe the intrinsic angular momentum of particles is another example of the importance of special orthogonal groups. Indeed, the spin group Spin(N) is a double cover of the special orthogonal group SO(N), i.e., Spin(N) is locally isomorphic to SO(N) (see, e.g., Ref. [1]).

In particle physics, the use of special orthogonal groups SO(N) have been very productive in the construction of Grand

*Corresponding author
Email addresses: nuno.cardoso@tecnico.ulisboa.pt (Nuno Cardoso), david.costa@tecnico.ulisboa.pt (David Emmanuel-Costa), nunogon@deec.uc.pt (Nuno Gonçalves), csimoes@ulg.ac.be (C. Simões)

Unified Theories (GUTs). The original idea of GUT models is to embed the Standard Model (SM) gauge group $\mathsf{SU}(3)_c \times \mathsf{SU}(2)_L \times \mathsf{U}(1)_Y$ in a larger simple Lie group, so that the three SM gauge couplings unify into a unique coupling. The first GUT model was proposed by Georgi and Glashow [2] in 1974 and it introduced $\mathsf{SU}(5)$ as the unifying gauge group. The group $\mathsf{SU}(5)$ has rank 4 as the SM group and the observed fermions are grouped in two unique representations $\bar{5}$ and 10, per generation.

The possibility of having a GUT model based on the special orthogonal group $\mathsf{SO}(10)$ was first accounted by Georgi [3, 4] and Fritzsch and Minkowski [5]. The $\mathsf{SO}(10)$ model brought new interesting features over $\mathsf{SU}(5)$. Each generation of SM fermions are accommodated in a unique 16-spinorial representation of $\mathsf{SO}(10)$ with an additional place for a singlet Weyl field, that can be interpreted later as a right-handed neutrino. These sterile neutrino states allow naturally to explain the observed oscillations of neutrinos through the Seesaw mechanism [6–10]; giving an extremely light mass to the active neutrinos when the sterile neutrino mass is of order of the unification scale. The $\mathsf{SO}(10)$ gauge interactions conserve parity thus making parity a continuous symmetry. Due to the fact that the rank of $\mathsf{SO}(10)$ is 5, there is an extra diagonal generator with quantum number $B - L$ as in the left-right symmetric models and it is indeed the minimal left-right symmetric GUT model. Finally, GUT models based on $\mathsf{SO}(N)$, apart from $\mathsf{SO}(6)$, turn out to be automatically free of gauge anomalies [11].

Since the appearance of first $\mathsf{SO}(10)$ GUT model, many models based on $\mathsf{SO}(10)$ have been proposed in the literature (cf. Refs. [12–22] and references therein). In addition, other models were implemented within $\mathsf{SO}(N)$ unification with a rank greater than 5, e.g., $\mathsf{SO}(12)$ [23], $\mathsf{SO}(14)$ [24], and $\mathsf{SO}(18)$ [25–27]. $\mathsf{SO}(18)$ turns out to be the minimal special orthogonal group that accommodates the three SM fermionic generations in a unique spinorial representation 256 by choosing properly the breaking chain down to the SM. There are also applications of $\mathsf{SO}(N)$ as unifying group in the context of models with extra-dimensions, e.g., $\mathsf{SO}(10)$ in 5D [28, 29], in orbifold 5D [30] and 6D [30–36]. The group $\mathsf{SO}(11)$ was also used in the context of Randall-Sundrum warped space [37, 38].

The breaking of a GUT $\mathsf{SO}(N)$ model down to the SM can be achieved by different breaking path, with possibly some intermediate mass scales. In order to understand the possible $\mathsf{SO}(N)$ breaking paths, it is important to identify its maximal subgroup (with the same rank as the higher group), so that one can express representations of $\mathsf{SO}(N)$ in terms of representations of the maximal subgroup and therefore understand the necessary Higgs sector. In particular, for the group $\mathsf{SO}(10)$ one identifies two important maximal subgroups [13, 16], namely $\mathsf{SU}(5) \times \mathsf{U}(1)$ and $\mathsf{SO}(6) \times \mathsf{SO}(4)$, which is equivalent to $\mathsf{SU}(4) \times \mathsf{SU}(2)_L \times \mathsf{SU}(2)_R$. The first subgroup can be broken into the usual $\mathsf{SU}(5)$. Instead, the second subgroup can be broken into the Pati-Salam model, $\mathsf{SU}(3) \times \mathsf{SU}(2)_L \times \mathsf{SU}(2)_R \times \mathsf{U}(1)_{B-L}$, in which the $B - L$ symmetry of the SM is gauged. It is worth to point out that one can also break $\mathsf{SO}(10)$ to the flipped-$\mathsf{SU}(5)$ [16], where the SM hypercharge is identified with a linear combination of the diagonal generator of $\mathsf{SU}(5)$ with extra $\mathsf{U}(1)$ generator of $\mathsf{SO}(10)$.

The purpose of this paper is to introduce the SO2pin library implemented in the C++ programming language. The idea behind the conception of SO2pin is the decomposition of Yukawa interactions invariant under $\mathsf{SO}(2N)$ in terms of $\mathsf{SU}(N)$ degrees of freedom. This decomposition is particularly useful for GUT models based on $\mathsf{SO}(2N)$ that break to an intermediate threshold symmetric under $\mathsf{SU}(N)$, since it allows to relate the Yukawa couplings in the intermediate theory with the GUT Yukawa couplings from the GUT theory, and thus leading to predictions. In general, this decomposition can be fastidious and error-prone. Our library is meant to simplify this task.

The SO2pin library relies on the oscillator expansion formalism, where the $\mathsf{SO}(N)$ spinor are written in an $\mathsf{SU}(N)$ basis realised through the introduction of creation and annihilation operators of a Grassmann algebra [39]. These operators and their algebra are simulated in SO2pin by means of doubly-linked lists as the appropriate data structure for these problems. This type of data structure has higher performance power, since it optimises the memory usage for long chains of operators and the data itself in memory do not need to be adjacent. Although the SO2pin library was projected with the groups $\mathsf{SO}(2N)$ in mind, it can be easily adapted to the groups $\mathsf{SO}(2N + 1)$ or even to other systems where creation and annihilation operators can be defined.

The paper is organised as follows. In the next section, we discuss the spinorial representations of $\mathsf{SO}(2N)$ in a basis in terms of the degrees of freedom of $\mathsf{SU}(N)$, through creation and annihilation operators defined in a Grassmann algebra. We then apply this method to decompose Yukawa interactions invariant under $\mathsf{SO}(2N)$ in terms of $\mathsf{SU}(N)$ interactions. In Section 3, we present the general structure of the SO2pin library, giving in detail the general functions and specific functions for $\mathsf{SO}(10)$. In Section 4, we explain the installation of our library and we show how to write simple programs. Then in Section 5, we give complete examples for computing Yukawa terms in $\mathsf{SO}(4)$ and $\mathsf{SO}(10)$ with the SO2pin library. Finally, we draw our conclusions in Section 6.

2. The SO($2N$) spinor representation

We review in this section the oscillator expansion technique [15, 39, 40] that is implemented in the SO2pin library. This technique has been actively explored for explicit computations of $\mathsf{SO}(10)$ Yukawa couplings [41–43]. The main idea of this technique is to write the two spinor representations of $\mathsf{SO}(2N)$ in a basis where the spinor components are expressed explicitly in terms of $\mathsf{SU}(N)$ fields. This is achieved by constructing a Grassmann algebra of creation and annihilation operators. One could have used a completely group theoretical approach as done in Ref. [44], but the oscillator expansion technique is more field theoretical and seems more intuitive to consider the case where the breaking of $\mathsf{SO}(2N)$ is done down to $\mathsf{SU}(N)$. In addition there are other methods in the literature [45–48] that can be used for computing the $\mathsf{SO}(2N)$ invariant couplings, but we shall not consider these methods in this paper.

We start by introducing the general properties of any special orthogonal group SO(N), which are the simple Lie group of all orthogonal $N \times N$ matrices $\mathbf{O}$ such that

$$\mathbf{O}^{\intercal}\mathbf{O} \;=\; \mathbf{O}\,\mathbf{O}^{\intercal} \;=\; \mathbf{1}\,, \tag{1}$$

with the special condition $|\mathbf{O}| = 1$. This group leaves invariant the bilinear

$$\mathbf{x}^{\intercal}\,\mathbf{y} \;=\; x_1\,y_1 \;+\; x_2\,y_2 \;+\; \cdots \;+\; x_N\,y_N\,, \tag{2}$$

when the N-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$ transform as

$$x_\mu \longrightarrow x'_\mu = O_{\mu\nu}\,x_\nu\,, \quad y_\mu \longrightarrow y'_\mu = O_{\mu\nu}\,y_\nu\,. \tag{3}$$

Making an infinitesimal group transformation, the matrix elements $O_{\mu\nu}$ can be expanded as

$$O_{\mu\nu} \;=\; \delta_{\mu\nu} \;-\; \frac{i}{2}\,\omega_{\mu\nu}M_{\mu\nu} \;+\; O(\omega^2)\,, \tag{4}$$

where $\omega_{\mu\nu}$ is a real antisymmetric tensor, while $M_{\mu\nu}$ are $N(N-1)/2$ independent $N \times N$-matrix generators of SO(N). In the vector representation, the generators are hermitian, $M^{\dagger}_{\mu\nu} = M_{\mu\nu}$, and they can be written as

$$\left(M_{\mu\nu}\right)_{mn} \;=\; i\left(\delta_{\mu m}\delta_{\nu n} - \delta_{\nu m}\delta_{\mu n}\right)\,, \tag{5}$$

implying $\mathrm{Tr}\,M_{\mu\nu} = 0$, and they satisfy the Lie algebra of SO(N) as

$$\left[M_{\mu\nu},\,M_{\rho\eta}\right] = i\left(\delta_{\mu\eta}M_{\nu\rho} - \delta_{\mu\rho}M_{\nu\eta} - \delta_{\nu\eta}M_{\mu\rho} + \delta_{\nu\rho}M_{\mu\eta}\right). \tag{6}$$

Within the Cartan classification, the Lie algebra associated to the group SO($2N+1$) is B_N while to SO($2N$) is D_N, with N being identified as the rank of the algebra. We focus now our discussion only on even-dimensional special groups SO($2N$). Note that the oscillator expansion technique can also be applied to the spinor representation of SO($2N+1$).

The spinor representations of SO($2N$) can be constructed if one introduces a set of matrices $\{\Gamma_\mu\}$, with $\mu = 1,\ldots N$, such that

$$x_1\,\Gamma_1^2\,y_1 \;+\; x_2\,\Gamma_2^2\,y_2 \;+\; \cdots \;+\; x_N\,\Gamma_N^2\,y_N \;=\; \mathbf{x}^{\intercal}\,\mathbf{y}\,. \tag{7}$$

In order to verify Eq. (7), one must necessarily impose that the matrices Γ_μ should obey to:

$$\{\Gamma_\mu, \Gamma_\nu\} = 2\delta_{\mu\nu}\,, \tag{8}$$

which form a Clifford algebra. It is straightforward to see that any ordered product of distinct gamma matrices gives rise to a complete set of linearly independent matrices. This fact leads to the construction of the so-called spinor representation of SO($2N$). In Appendix A we give a general proof of the existence of the matrices Γ_μ. In fact, for any even-dimensional Clifford algebra there is only one irreducible representation of dimension 2^N. Instead of writing explicitly the matrices Γ_μ via the $2^N \times 2^N$ generalised Dirac matrices formed from the direct product of the Pauli matrices, we write them in terms of a set of creation ($b_i^{\dagger}$) and annihilation (b_i) operators acting on the Hilbert space as

$$\{b_i, b_j^{\dagger}\} = \delta_{ij}\,, \quad \{b_i, b_j\} = 0 = \{b_i^{\dagger}, b_j^{\dagger}\}\,, \tag{9}$$

with $i \;=\; 1, ..., N$. Each pair b_i, $b_i^{\dagger}$ of operators can be constructed directly from linear combinations of pairs of Γ-matrices as

$$b_j = \frac{1}{2}(i\,\Gamma_{2j-1} \;+\; \Gamma_{2j})\,, \quad b_j^{\dagger} = \frac{1}{2}(-i\,\Gamma_{2j-1} \;+\; \Gamma_{2j}), \tag{10}$$

with the inverted relation given by

$$\Gamma_{2j-1} = -i(b_j - b_j^{\dagger})\,, \quad \Gamma_{2j} = (b_j + b_j^{\dagger}), \tag{11}$$

showing a one-to-one correspondence. General formulae for the correspondence between the Clifford and the Grassmann algebrae are found in Appendix B.

The advantage of this approach is that one does not need to write explicitly the operators b_i, $b_i^{\dagger}$, one needs only to define the vacuum state $|0\rangle$. One defines the Fock vacuum as the vector $|0\rangle = |0, 0, 0, ..., 0\rangle$ corresponding to N unoccupied states, which is defined by

$$b_i\;|0\rangle \;=\; 0\,, \text{for all } i = 1, ..., N\,. \tag{12}$$

One-state vector can then be represented as

$$b_i^{\dagger}\;|0\rangle \;=\; |0, 0, ..., 1, ..., 0\rangle\;, \tag{13}$$

where the non-zero entry is at position i. We have just derived the building blocks to construct the spinor representation of SO($2N$) in terms of states obtained from the action of the creation operators. Moreover, defining the set of operators $T_{ij} \equiv b_i^{\dagger}\,b_j$, it is easy to verify that they satisfy the algebra of U(N) as

$$\left[T_{ij},\,T_{kl}\right] \;=\; \delta_{kj}T_{il} \;-\; \delta_{il}T_{kj}\,. \tag{14}$$

It is then not surprising to observe that the basis of vectors obtained through the action of products of creation operators on the Fock vacuum,

$$b_{i_1}^{\dagger}\,b_{i_2}^{\dagger}\,...\,b_{i_p}^{\dagger}\;|0\rangle\;, \quad i_1 < i_2 < \cdots < i_p, \tag{15}$$

expands any vector $|\Psi\rangle$ with coefficients being irreducible fully-antisymmetric U(N) tensors, $\psi^{i_1\ldots i_p}$. This fact allows us to write the spinor representations of SO($2N$) in terms of irreducible SU(N) tensors.

2.1. *Decomposition into the* SU(N) *basis*

The general expression for the spinor representation of SO($2N$) written in terms of the SU(N) fields is given by

$$|\Psi\rangle \;=\; \sum_{p=0}^{N}\;\frac{1}{p!}\;b_{i_1}^{\dagger}\,...\,b_{i_p}^{\dagger}\;|0\rangle\;\psi^{i_1\ldots i_p}\,. \tag{16}$$

The completely antisymmetric tensors $\psi^{i_1\ldots i_p}$ have dimension $\binom{N}{p}$. An easy way to compute the dimension of all tensors in

Eq. (16) is by noting that it can be read from the Nth-row of the Tartaglia's triangle[1]. For tensors with large number of indices it may be convenient to reduce them with help of their conjugate tensors using the Levi-Civita invariant tensor of dimension N,

$$\overline{\psi}_{i_{p+1}\ldots i_N} = \frac{1}{p!}\,\varepsilon_{i_1\ldots i_N}\,\psi^{i_1\ldots i_p}\,, \tag{17}$$

and therefore Eq. (16) becomes

$$\begin{aligned}|\Psi\rangle \;=\; &|0\rangle\,\psi \;+\; b_i^\dagger\,|0\rangle\,\psi^i \;+\; \frac{1}{2}\,b_i^\dagger b_j^\dagger\,|0\rangle\,\psi^{ij}\\ &+\cdots\\ &+\;\frac{\varepsilon^{i_1 i_2\cdots i_N}}{2!(N-2)!}\,b_{i_1}^\dagger b_{i_2}^\dagger\cdots b_{i_{N-2}}^\dagger\,|0\rangle\,\bar{\psi}_{i_{N-1}i_N}\\ &+\;\frac{\varepsilon^{i_1 i_2\cdots i_N}}{(N-1)!}\,b_{i_1}^\dagger b_{i_2}^\dagger\cdots b_{i_{N-1}}^\dagger\,|0\rangle\,\bar{\psi}_{i_N}\\ &+\;b_1^\dagger b_2^\dagger\cdots b_N^\dagger\,|0\rangle\,\bar{\psi}\,.\end{aligned} \tag{18}$$

The dimension of the vector space in Eq. (18) is 2^N which is in agreement with the dimension of the Γ-matrices. Within the $\mathsf{SU}(N)$ basis, given by the vectors of Eq. (15), any spinor $|\Psi\rangle$ corresponds to a column vector Ψ,

$$\Psi = \begin{pmatrix}\psi & \psi^j & \psi^{jk} & \cdots & \overline{\psi}_{jk} & \overline{\psi}_j & \overline{\psi}\end{pmatrix}^{\mathsf{T}}\,. \tag{19}$$

In this spinor representation of dimension 2^N, the states Ψ transform under $\mathsf{SO}(2N)$ as

$$\Psi \;\longrightarrow\; \Psi' = U(\omega)\,\Psi \quad \text{or} \quad |\Psi'\rangle = U(\omega)\,|\Psi\rangle\,, \tag{20}$$

where the unitary transformation $U(\omega)$ is given by

$$U(\omega) = \exp\left(-\frac{i}{2}\omega_{\mu\nu}\Sigma_{\mu\nu}\right). \tag{21}$$

The generators $\Sigma_{\mu\nu}$ of the spinor representation are constructed in terms of the Γ-matrices as

$$\Sigma_{\mu\nu} = \frac{1}{2i}\left[\Gamma_\mu,\,\Gamma_\nu\right], \tag{22}$$

with $\Sigma_{\mu\nu}^\dagger = \Sigma_{\mu\nu}$ and $\mathrm{Tr}\,\Sigma_{\mu\nu} = 0$, which guaranties the unitarity of $U(\omega)$ and $|U(\omega)| = 1$, respectively. It is straightforward to verify that $\Sigma_{\mu\nu}$ satisfies the algebra of $\mathsf{SO}(2N)$ given in Eq. (6). It turns out that the spinor representation $|\Psi\rangle$ with dimension 2^N given in Eq. (18) is in fact reducible. This fact can easily be demonstrate by observing that the product of $2N$ Γ-matrices, Γ_0, defined as

$$\Gamma_0 \;=\; i^N\Gamma_1\,\Gamma_2\,\ldots\,\Gamma_{2N}\,, \tag{23}$$

anticommutes with all Γ_μ matrices, but it commutes with $\Sigma_{\mu\nu}$ and therefore splits the spinor Ψ into two nonequivalent irreducible spinors Ψ_+ and Ψ_- of dimension $2^{N/2}$, given by

$$\Psi_\pm = \frac{1}{2}(1\pm\Gamma_0)\Psi\,. \tag{24}$$

[1]This mathematical representation is also known as the Pascal's triangle. The triangle was already known centuries before in China, India and Iran.

The projectors $\frac{1}{2}(1\pm\Gamma_0)$ are in total analogy with the chiral projectors known in the Dirac space (see projector properties in Appendix A). The chiral states Ψ_+ are generated by the action of an even number of creation operators on the vacuum state $|0\rangle$, while the chiral states Ψ_- are generated by the action of an odd number of creation operators. Observing Eq. (18) one obtains

$$\Psi_+ = \begin{pmatrix}\psi\\ \psi^{ij}\\ \overline{\psi}_i\\ \vdots\end{pmatrix}, \quad \Psi_- = \begin{pmatrix}\overline{\psi}\\ \overline{\psi}_{ij}\\ \psi^i\\ \vdots\end{pmatrix}, \tag{25}$$

and one can distinguish two cases in $\mathsf{SO}(2N)$: when N is an odd integer Ψ_+ and Ψ_- are self-conjugate, while when N is even Ψ_+ and Ψ_- are distinct spinor representations.

2.2. *Yukawa interactions*

In this subsection, we sketch the construction of Yukawa interactions in a generic GUT model ruled by $\mathsf{SO}(2N)$ expressed in terms of $\mathsf{SU}(2N)$ tensor fields. This is particularly useful when the group $\mathsf{SO}(2N)$ is broken to its $\mathsf{SU}(2N)$ subgroup at some intermediate scale, since below the breaking scale one gets new relations among the Yukawa couplings of the $\mathsf{SU}(N)$ theory. In what follows, we shall assume that the fermionic degrees of freedom belong to irreducible $\mathsf{SO}(2N)$ spinor representations and the Higgs fields transform as complete antisymmetric tensors of $\mathsf{SO}(2N)$. Since the GUT gauge group commutes with space-time symmetries, it is convenient to write the fermionic degrees of freedom in terms of left-handed Weyl fields. In order to write the most general Yukawa interaction invariant under $\mathsf{SO}(2N)$, it is useful to express the transposition of $U(\omega)$ in terms of its corresponding inverse matrix $U^\dagger(\omega)$ as

$$U^{\mathsf{T}}(\omega) = B\,U^\dagger(\omega)\,B^{-1}\,, \tag{26}$$

through the matrix B, which has the property

$$B^{-1}\,\Gamma_\mu^{\mathsf{T}}\,B \;=\; \Gamma_\mu\,. \tag{27}$$

In Appendix A, it is shown that such operator B always exists and it can be written as

$$B = \prod_{\mu=odd}\Gamma_\mu = (-i)^N\prod_{k=1}^{N}(b_k - b_k^\dagger)\,. \tag{28}$$

From the above equation, one sees that the operator B anticommutes with Γ_0 when N is odd, while instead it commutes when N is even. When simplifying expressions involving the operator B, it turns out to be more convenient to write it with contracted indices as

$$B = \frac{(-i)^N}{N!}\,\varepsilon^{jk\cdots z}\,(b_j - b_j^\dagger)(b_k - b_k^\dagger)\cdots(b_z - b_z^\dagger)\,, \tag{29}$$

where $\varepsilon^{jk\cdots z}$ is the Levi-Civita antisymmetric tensor with N indices. From Eq. (26), one deduces that the combination $\Psi^{\mathsf{T}}B$ does transform as

$$\Psi^{\mathsf{T}}B \;\longrightarrow\; \Psi'^{\mathsf{T}}B = \Psi^{\mathsf{T}}B\,U^\dagger(\omega)\,, \tag{30}$$

and one concludes that for any pair of spinors Ψ_1 and Ψ_2 the bilinear $\Psi_1^\intercal BC^{-1}\Psi_2$ is invariant under SO($2N$). Due to the fact that Ψ_1 and Ψ_2 are assumed as fermionic fields, the presence of charge conjugation matrix C ensures that this bilinear is also invariant under Lorentz transformations. Furthermore, the matrices Γ_μ transform as

$$U(\omega)^\dagger\,\Gamma_\mu\,U(\omega)\;=\;O_{\mu\nu}\,\Gamma_\nu\,, \tag{31}$$

where the matrix O is given in Eq. (4). Using this result one can write an SO($2N$) invariant Yukawa coupling combining the fermion in a spinor representation Ψ with the Higgs scalar ϕ_μ, that transforms like a vector according to Eq. (3):

$$\Psi^\intercal B\,C^{-1}\Gamma_\mu\,\Psi\,\phi_\mu\,. \tag{32}$$

The relation given in Eq. (31) can be generalised to any product of Γ-matrices as

$$U(\omega)^\dagger\,\Gamma_\mu\Gamma_\nu\cdots\Gamma_\rho\,U(\omega)=O_{\mu\mu'}O_{\nu\nu'}\cdots O_{\rho\rho'}\Gamma_{\mu'}\Gamma_{\nu'}\cdots\Gamma_{\rho'}\,. \tag{33}$$

This general formula allows us to write the most general gauge-invariant Yukawa coupling under SO($2N$). In the language of creation and annihilation operators one writes

$$Y_{ab}\,\left\langle\Psi^*_{k\,a}\right|B\,C^{-1}\,\Gamma_{[\mu_1}\Gamma_{\mu_2}\,...\,\Gamma_{\mu_m]}\,\left|\Psi_{l\,b}\right\rangle\,\phi_{\mu_1\mu_2\cdots\mu_m}\,, \tag{34}$$

where the indices $k,l=+,-$ denote the two possible irreducible spinors; a and b are flavor indices and the elements Y_{ab} of the Yukawa matrix; $\phi_{\mu_1\mu_2\cdots\mu_m}$ with $m=1,\ldots N$ is a scalar tensor fully antisymmetric. Due to the fact that $\phi_{\mu_1\mu_2\cdots\mu_m}$ is fully antisymmetric, any Γ-matrix product in the formula in Eq. (34) should also be made fully antisymmetric, and therefore one has

$$\Gamma_{[\mu_1}\Gamma_{\mu_2}...\Gamma_{\mu_m]}\;=\;\frac{1}{m!}\sum_P(-1)^{\delta P}\Gamma_{\mu_{P(1)}}\Gamma_{\mu_{P(2)}}\cdots\Gamma_{\mu_{P(m)}}\,, \tag{35}$$

where the sum runs over the permutations and δP takes 0 for even number of permutations and 1 for odd number of permutations. We notice that the general formula given in Eq. (34) can also be applied to the case where the SO($2N$) spinors are taken as scalar fields yielding a pure scalar interaction in the scalar potential. In this case, the bracket in Eq. (34) should not include the charge conjugation matrix C.

In some cases, the bracket given in Eq. (34) vanishes automatically. This can be well understood by taking into account the general properties of the projectors $\frac{1}{2}(1\pm\Gamma_0)$ and the fact that Γ_0 commutes with an even number of Γ-matrix product or anticommutes with an odd number. Thus, in the case $N+m$ is an odd number, it implies

$$\left\langle\Psi^*_{k\,a}\right|B\,C^{-1}\,\Gamma_{\mu_1}\Gamma_{\mu_2}\cdots\Gamma_{\mu_m}\,\left|\Psi_{k\,a}\right\rangle\;=\;0\,, \tag{36}$$

while in the case $N+m$ is an even number, it implies

$$\left\langle\Psi^*_{k\,a}\right|B\,C^{-1}\,\Gamma_{\mu_1}\Gamma_{\mu_2}\cdots\Gamma_{\mu_m}\,\left|\Psi_{l\,b}\right\rangle\;=\;0\,, \tag{37}$$

when $k\neq l$.

Concerning the antisymmetric tensor $\phi_{\mu_1\mu_2\cdots\mu_m}$ some comments are in order. There are N distinct fully antisymmetric tensors with dimension $\binom{2N}{m}$ with $1\leq m\leq N$[2]. Moreover, the representation with dimension $\binom{2N}{N}$, denoted as $\Delta_{\mu_1\mu_2\cdots\mu_N}$, is indeed a reducible representation that can be decomposed into two irreducible representations [12, 14]. For N odd into two irreducible pairs self-conjugate representations of dimension $\frac{1}{2}\binom{2N}{N}$:

$$\Delta_{\mu_1\mu_2\cdots\mu_N}=\overline{\phi}_{\mu_1\mu_2\cdots\mu_N}\;+\;\phi_{\mu_1\mu_2\cdots\mu_N}\,, \tag{38}$$

where

$$\begin{pmatrix}\overline{\phi}_{\mu_1\cdots\mu_N}\\ \phi_{\mu_1\cdots\mu_N}\end{pmatrix}\equiv\frac{1}{2}\begin{pmatrix}\delta_{\mu_1\nu_1}\cdots\delta_{\mu_N\nu_N}+\frac{i}{N!}\varepsilon_{\mu_1\cdots\mu_N\nu_1\cdots\nu_N}\\ \delta_{\mu_1\nu_1}\cdots\delta_{\mu_N\nu_N}-\frac{i}{N!}\varepsilon_{\mu_1\cdots\mu_N\nu_1\cdots\nu_N}\end{pmatrix}\Delta_{\nu_1\cdots\nu_N}\,. \tag{39}$$

Instead, for N even, one has the following decomposition

$$\Delta_{\mu_1\mu_2\cdots\mu_N}=\phi^+{}_{\mu_1\mu_2\cdots\mu_N}\;+\;\phi^-_{\mu_1\mu_2\cdots\mu_N}\,, \tag{40}$$

where

$$\begin{pmatrix}\phi^+_{\mu_1\cdots\mu_N}\\ \phi^-_{\mu_1\cdots\mu_N}\end{pmatrix}\equiv\frac{1}{2}\begin{pmatrix}\delta_{\mu_1\nu_1}\cdots\delta_{\mu_N\nu_N}+\frac{1}{N!}\varepsilon_{\mu_1\cdots\mu_N\nu_1\cdots\nu_N}\\ \delta_{\mu_1\nu_1}\cdots\delta_{\mu_N\nu_N}-\frac{1}{N!}\varepsilon_{\mu_1\cdots\mu_N\nu_1\cdots\nu_N}\end{pmatrix}\Delta_{\nu_1\cdots\nu_N}\,. \tag{41}$$

When computing the full expression given by Eq. (34) for the maximal number of Γ-matrices, one verifies that only one of the irreducible components of $\Delta_{\mu_1\,...\,\mu_N}$ couples to the Yukawa term. Indeed, for N odd, one has $k=l$ and therefore only $\overline{\phi}_{\mu_1\cdots\mu_N}(\phi_{\mu_1\cdots\mu_N})$ couples to the Yukawa when $k=+(-)$, otherwise, for N even, one has $k\neq l$ and therefore only $\phi^+_{\mu_1\cdots\mu_N}(\phi^-_{\mu_1\cdots\mu_N})$ couples to the Yukawa when $k=-(+)$. One then concludes that the computation of the Yukawa given by Eq. (34) can be performed directly using the reducible representation $\Delta_{\mu_1\mu_2\cdots\mu_N}$ as

$$Y_{ab}\,\left\langle\Psi^*_{k\,a}\right|B\,C^{-1}\,\Gamma_{[\mu_1}\Gamma_{\mu_2}\,...\,\Gamma_{\mu_N]}\,\left|\Psi_{l\,b}\right\rangle\,\Delta_{\mu_1\,...\,\mu_N}\,, \tag{42}$$

without the loss of generality.

For illustrative purpose, the antisymmetric tensor representations have the following dimension in the case of SO(10): $\phi_\mu\sim\mathbf{10}$, $\phi_{\mu\nu}\sim\mathbf{45}$, $\phi_{\mu\nu\lambda}\sim\mathbf{120}$, $\phi_{\mu\nu\lambda\sigma}\sim\mathbf{210}$, $\phi_{\mu\nu\lambda\sigma\gamma}\sim\mathbf{126}$, and $\overline{\phi}_{\mu\nu\lambda\sigma\gamma}\sim\overline{\mathbf{126}}$.

One can also express the tensor $\phi_{\mu\nu\lambda...\sigma}$ in terms of SU(N) tensors. This can be easily computed by expanding the quantity $\Gamma_\mu\Gamma_\nu\Gamma_\lambda...\Gamma_\sigma\,\phi_{\mu\nu\lambda...\sigma}$ in terms of the creation and annihilation operators [41] as

$$\begin{aligned}\Gamma_\mu\Gamma_\nu\Gamma_\lambda...\Gamma_\sigma\phi_{\mu\nu\lambda...\sigma}\;=\;&b_i^\dagger b_j^\dagger b_k^\dagger...b_n^\dagger\phi_{c_ic_jc_k...c_n}\\&+(b_ib_j^\dagger b_k^\dagger...b_n^\dagger\phi_{\bar{c}_ic_jc_k...c_n}+\text{perms})\\&+(b_ib_jb_k^\dagger...b_n^\dagger\phi_{\bar{c}_i\bar{c}_jc_k...c_n}+\text{perms})+...\\&+(b_ib_jb_k...b_{n-1}b_n^\dagger\phi_{\bar{c}_i\bar{c}_k...\bar{c}_{n-1}c_n}+\text{perms})\\&+(b_ib_jb_k...b_n\phi_{\bar{c}_i\bar{c}_j\bar{c}_k...\bar{c}_n})\,,\end{aligned} \tag{43}$$

[2] Representations of dimension $\binom{2N}{m}$ with $m>N$ are equivalent to representations with dimension $\binom{2N}{m-N}$.

where $\phi_{\dots c_j \dots} \equiv \phi_{\dots 2j \dots} + i\,\phi_{\dots 2j-1 \dots}$ and $\phi_{\dots \bar{c}_j \dots} \equiv \phi_{\dots 2j \dots} - i\,\phi_{\dots 2j-1 \dots}$. The new tensors in equation Eq. (43) manifest completely antisymmetry, i.e.,

$$\begin{aligned}
\phi_{\dots c_i \dots c_j \dots} &= -\phi_{\dots c_j \dots c_i \dots}\,, \\
\phi_{\dots \bar{c}_i \dots \bar{c}_j \dots} &= -\phi_{\dots \bar{c}_j \dots \bar{c}_i \dots}\,, \\
\phi_{\dots c_i \dots \bar{c}_j \dots} &= -\phi_{\dots \bar{c}_j \dots c_i \dots}\,.
\end{aligned} \tag{44}$$

In Appendix C, we compile all the antisymmetric tensors of SO(10) explicitly written in terms of SU(5) representations.

2.3. Methods and rules

We compile in this section the rules that are the basis for defining the behaviour of the SO2pin library. In order to concretise the use of the rules, let us first take a simple bracket containing a set of annihilation and creator operators,

$$\Upsilon^{ijl}_{kmn} = \langle 0|\, b_i\, b_j\, b^\dagger_k\, b_l\, b^\dagger_m\, b^\dagger_n\, |0\rangle\,. \tag{45}$$

The tensor Υ^{ijl}_{kmn} differentiates upper and lower indices associated with annihilation and creation operator indices, respectively. This distinction is important to obtain a final expression with a index structure consistent with SU(N). The computation of the bracket from Eq. (45) relies on the use of the relation given in Eqs. (9) and (12). We present two different strategies, that we call *normal ordering* and *reverse ordering* methods. We first discuss the reverse ordering method.

Reverse ordering method

In this case, we move according to Eq. (9) either the annihilation operators forward to the right until it cancels with $|0\rangle$, $b_i\,|0\rangle$=0, or the creation operators backward to the left to cancel with $\langle 0|$, i.e., $\langle 0|\, b^\dagger_i = 0$. Hence, moving the operator b_l to the right, we have

$$\begin{aligned}
&\langle 0|\, b_i\, b_j\, b^\dagger_k\, b_l\, b^\dagger_m\, b^\dagger_n\, |0\rangle = \\
&= \langle 0|\, b_i\, b_j\, b^\dagger_k \left(\delta_{lm} - b^\dagger_m b_l\right) b^\dagger_n\, |0\rangle \\
&\quad \dots \\
&= \delta_{lm}\, \langle 0|\, b_i\, b_j\, b^\dagger_k\, b^\dagger_n\, |0\rangle - \delta_{ln}\, \langle 0|\, b_i\, b_j\, b^\dagger_k\, b^\dagger_m\, |0\rangle\,.
\end{aligned} \tag{46}$$

We complete the computation of $\langle 0|\, b_i\, b_j\, b^\dagger_k\, b^\dagger_n\, |0\rangle$ and $\langle 0|\, b_i\, b_j\, b^\dagger_k\, b^\dagger_m\, |0\rangle$ using the same procedure, obtaining

$$\begin{aligned}
\langle 0|\, b_i\, b_j\, b^\dagger_k\, b^\dagger_n\, |0\rangle &= \delta_{jk}\,\delta_{in} - \delta_{jn}\,\delta_{ik}\,, \\
\langle 0|\, b_i\, b_j\, b^\dagger_k\, b^\dagger_m\, |0\rangle &= \delta_{jk}\,\delta_{im} - \delta_{jm}\,\delta_{ik}\,.
\end{aligned} \tag{47}$$

The final result for $\langle 0|\, b_i\, b_j\, b^\dagger_k\, b_l\, b^\dagger_m\, b^\dagger_n\, |0\rangle$, in terms of δ's, is then given by

$$\delta_{lm}(\delta_{jk}\,\delta_{in} - \delta_{jn}\,\delta_{ik}) - \delta_{ln}(\delta_{jk}\,\delta_{im} - \delta_{jm}\,\delta_{ik})\,. \tag{48}$$

As already said, instead we move the annihilation operators to the right-handed side to cancel at $|0\rangle$, we can move the creation operators to the left to cancel when reach $\langle 0|$. However, once we choose to move either annihilation or creation operators, we need to maintain this choice until the end of the computation.

Normal ordering method

In the normal ordering method, we use the relations in Eq. (9) to rearrange the creation and annihilation operators in such a way that all annihilation operators are on the left-handed side while all creation operators are on the right-handed side, as

$$\begin{aligned}
&\langle 0|\, b_i\, b_j\, b^\dagger_k\, b_l\, b^\dagger_m\, b^\dagger_n\, |0\rangle = \\
&= \langle 0|\, b_i\, b_j\, (\delta_{kl} - b_l\, b^\dagger_k)\, b^\dagger_m\, b^\dagger_n\, |0\rangle \\
&= \delta_{kl}\, \langle 0|\, b_i\, b_j\, b^\dagger_m\, b^\dagger_n\, |0\rangle - \langle 0|\, b_i\, b_j\, b_l\, b^\dagger_k\, b^\dagger_m\, b^\dagger_n\, |0\rangle\,.
\end{aligned} \tag{49}$$

Then we compute $\langle 0|\, b_i\, b_j\, b_l\, b^\dagger_k\, b^\dagger_m\, b^\dagger_n\, |0\rangle$ and $\langle 0|\, b_i\, b_j\, b^\dagger_m\, b^\dagger_n\, |0\rangle$ by using the relation,

$$\begin{aligned}
&\langle 0|\, b_{i_1} b_{i_2} \dots b_{i_k}\, b^\dagger_{j_1} b^\dagger_{j_2} \dots b^\dagger_{j_k}\, |0\rangle = \\
&\frac{1}{(N-k)!}\, \varepsilon^{i_1 i_2 \dots i_k l_{k+1} \dots l_N}\, \varepsilon_{j_k \dots j_2 j_1 l_{k+1} \dots l_N}\,,
\end{aligned} \tag{50}$$

that holds for SU(N) with $k \leq N$.

Up to now, we did not mention in which SU(N) framework we are computing the expression given in Eq. (45). If we choose to compute it in SU(5) (i.e., $N = 5$) the brackets in Eq. (49) take the values

$$\begin{aligned}
\langle 0|\, b_i\, b_j\, b^\dagger_m\, b^\dagger_n\, |0\rangle &= \frac{1}{3!}\, \varepsilon^{ij\alpha\beta\gamma}\, \varepsilon_{nm\alpha\beta\gamma}\,, \\
\langle 0|\, b_i\, b_j\, b_l\, b^\dagger_k\, b^\dagger_m\, b^\dagger_n\, |0\rangle &= \frac{1}{2!}\, \varepsilon^{ijl\alpha\beta}\, \varepsilon_{nmk\alpha\beta}\,,
\end{aligned} \tag{51}$$

and the tensor Υ^{ijl}_{kmn} defined in Eq. (45) becomes then

$$\Upsilon^{ijl}_{kmn} = \frac{1}{6}\,\delta^l_k\, \varepsilon^{ij\alpha\beta\gamma}\, \varepsilon_{nm\alpha\beta\gamma} - \frac{1}{2} \varepsilon^{ijl\alpha\beta}\, \varepsilon_{nmk\alpha\beta}. \tag{52}$$

We are ready to summarise all the rules mentioned above. It is known that a complete bracket expression is composed by creation and annihilation operators, and some other fields. The idea is to get rid of all operators by using the relations in Eq. (9) and use the terms arising from these calculations to simplify the remaining expression (e.g., fields if there are any). In order to have a consistent expression, each element in the bracket must obey certain rules, those that we list below for a general SO($2N$):

1. The number of creation operators in a bracket expression must be equal to the number of annihilation operators.
2. The number of contiguous creation (or annihilation) operators must be equal or less than N for SO($2N$).
3. The operators b_i inside the bracket expression are written on the left-handed side while the operators $b^\dagger_i$ are written on the right-handed side, otherwise the result is zero, i.e., $b_i\,|0\rangle = 0$ and $\langle 0|\, b^\dagger_i = 0$.
4. The difference between the number of upper and lower indices in the fields must be zero or multiple of N for any SO($2N$).

Within this framework, we shall give the *Hermitian conjugation* and the *transposition* operations on a general vector

$$\begin{aligned}
|\Psi\rangle &= |0\rangle\, \psi + b^\dagger_i\, |0\rangle\, \psi^i + \frac{1}{2} b^\dagger_i b^\dagger_j\, |0\rangle\, \psi^{ij} \\
&\quad + \frac{1}{12} \varepsilon^{ijklm} b^\dagger_k b^\dagger_l b^\dagger_m\, |0\rangle\, \psi_{ij} + \dots\,,
\end{aligned} \tag{53}$$

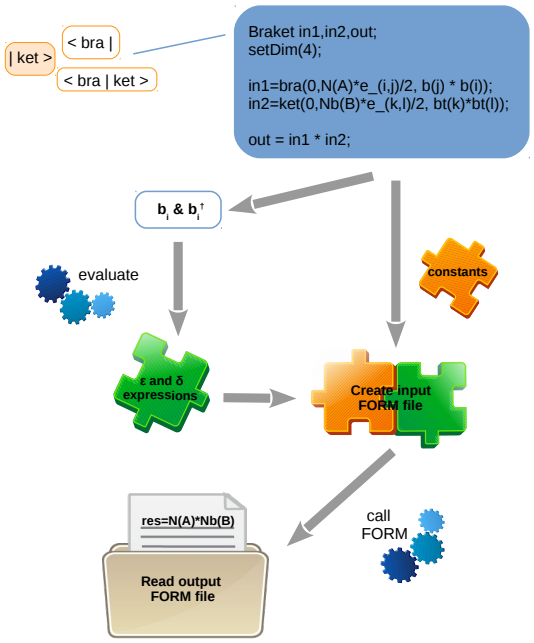

Figure 1: Scheme of SO2pin library exemplifying how to compute $\frac{\varepsilon_{ij}\varepsilon_{kl}}{4} N_A N_B \langle 0| b_j b_i b_k^\dagger b_l^\dagger |0\rangle$ in SO(4).

which are

$$\langle \Psi| = \psi^* \langle 0| + \psi_i \langle 0| b_i + \frac{1}{2}\psi_{ij} \langle 0| b_j b_i + \frac{1}{12}\psi^{ij}\varepsilon_{ijklm} \langle 0| b_m b_l b_k + \dots, \tag{54}$$

and

$$\langle \Psi^*| = \psi \langle 0| + \psi^i \langle 0| b_i + \frac{1}{2}\psi^{ij} \langle 0| b_j b_i + \frac{1}{12}\psi_{ij}\varepsilon^{ijklm} \langle 0| b_m b_l b_k + \dots, \tag{55}$$

respectively, where we make the usual identification of lower indices as $\psi_{ij} = \left(\psi^{ij}\right)^*$.

3. SO2pin, a C++ library

In this section we present the structure of the SO2pin library, that can be found in http://sospin.hepforge.org, comment on the data structure representation and give a list of the most important functions to use when writing a program linked to the SO2pin library. As mentioned before, the SO2pin code is entirely written in C++ and it is based on operations over creation and annihilation operators. We give in Fig. 1 the pictorial scheme to explain how a SO2pin program works. It works like this: the building blocks of the code, i.e., the $\langle bra|$ and the $|ket\rangle$ entities, can be defined either in a main file program (`in1` and `in2` in the picture) or included in some `include` file and called in the main file. The implemented operations in the SO2pin library are the following:

- $\langle bra| \cdot |ket\rangle$

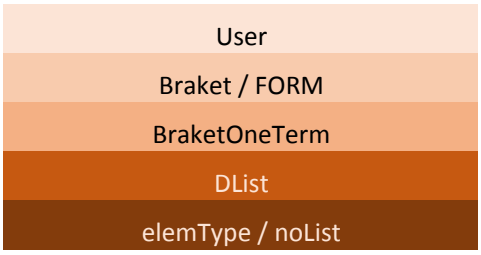

Figure 2: Hierarchy of the SO2pin library.

- $\langle bra| \cdot$ free
- free $\cdot |ket\rangle$
- $\langle bra| + \langle bra|$
- $|ket\rangle + |ket\rangle$
- free + free
- free · free
- $\langle bra|ket\rangle + \langle bra|ket\rangle$
- $\langle bra|ket\rangle \cdot \langle bra|ket\rangle$ [This operation is only allowed after the evaluation of the expression.]

Once the expression to evaluate is defined (`out` in the picture) the approach to solve it is the following: the expression to evaluate is split in two parts, one with all constants and another one containing operators; the operator part will be worked out using one of the two methods described in Section 2.1 leading to an intermediate expression written in terms of ε's or δ's. Then, the constant part and the intermediate expression are joined together to lead to a semi-final expression. If the starting example is simple, the result will be simple. On the other hand, if the example is somewhat complex, the expression obtained at this stage is rather large and needs extra simplifications for increasing the readability of the final expressions. Hence, in order to make the reading of all results as easier as possible, we include the possibility to simplify the expression obtained so far with the Symbolic Manipulation System FORM [49]. Once the constant part and the intermediate expression are joined together, it is created an input file to be run by FORM leading to a more simplified expression. The final expression is then read back from the FORM output file to the program. Note that, in the output FORM file, we first present all partial results and then the complete result at the end of the file.

We have opted to include the FORM program as a tool to do the final simplifications if needed. For the sake of curiosity we have implemented in FORM all the procedure used in SO2pin; however, the running time measured is far larger than in the case of using our library in C++. This shows the performance power of our choice for the appropriate data structure, which is described in the next section.

In Fig. 2 we define how the class structure works. In terms of level abstraction the low level implementation of the basic elements for the expression evaluation are the structures `elemType` and `noList`. They represent the expression elements or nodes. The class **DList** is then build over this abstraction level and represents a linked connection of nodes. Then **BraketOneTerm** and **Braket** classes represent complete expressions that can be evaluated and simplified by FORM and by the User at the higher abstraction level.

3.1. Data structure representation

In order to manipulate sequences of operators b_i or $b_i^\dagger$, we need to find the adequate data structure to store and further evaluate such sequences. Such data structures should require the following criteria:

- optimize memory usage - since the sequences can get extremely long;
- optimize flexibility of permutations - adjacency in memory is not relevant;
- standardize description of all elements - to ease interpretation and evaluation, and to reduce memory waste in contraction and expansion operations.

We have adopted the **doubly-linked list** scheme as the appropriate solution to the problem. A doubly-linked list consists on the list of connected nodes, which include specific data objects, such as each node is linked to previous and next nodes in the list. This is advantageous since one changes only the pointers without modifying the content and their position on the memory. The doubly-linked list scheme is implemented by a C++ class named as **DList**. The full method list is included in the Appendix D.

The different types of elements within a sequence are encoded as bit-fields of an integer type (**int**) with the purpose of optimising the memory usage. Thus, each **DList**-node can account for the operators b_i and $b_i^\dagger$, as well as the constants and the Kronecker symbol δ, with its indices. In Fig. 3, we illustrate the concept of the **DList** class for a simple sequence.

Computation Performance. In order to give an estimate of the performance of the computation, we measured the time consumed and the memory used in computing a sequence of creation and annihilation operators. We run the test in a x64 LINUX machine (Ubuntu) with an Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz.

We tested the following expression for SO(10),

$$\langle 0|\, b_{i_1} b_{i_2} b^\dagger_{j_1} b_{i_3} b_{i_4} b_{i_5} b^\dagger_{j_2} b^\dagger_{j_3} b_{i_6} b^\dagger_{j_4} b_{i_7} b_{i_8} b^\dagger_{j_5} b^\dagger_{j_6} b^\dagger_{j_7} b^\dagger_{j_8} \,|0\rangle \ , \tag{56}$$

the program needs a total of 3.57 MB and 0.0210 s to evaluate this expression in the delta form and 1.39 MB and 0.0044 s to evaluate the same expression to Levi-Civita tensor form.

Running the following sequence in SO(18),

$$\langle 0|\, b_{i_1} b_{i_2} b_{i_3} b_{i_4} b_{i_5} b_{i_6} b_{i_7} b_{i_8} b_{i_9} b^\dagger_{j_1} b^\dagger_{j_2} b^\dagger_{j_3} b^\dagger_{j_4} b^\dagger_{j_5} b^\dagger_{j_6} b^\dagger_{j_7} b^\dagger_{j_8} b^\dagger_{j_9} \,|0\rangle \ , \tag{57}$$

the program needs a total of 454.15 MB and 2.80 s to evaluate this expression in the delta form and 1.41 MB and 0.00064 s to evaluate the same expression to the Levi-Civita tensor form. The large amount of memory used to evaluate in the delta form is due to the number of terms generated in this way, a total of 9! = 362880 terms, while for the evaluation to Levi-Civita tensor the result has only one final term.

3.2. General functions

In this section we list the most general functions needed to write a program using SO2qi library; it is divide in three subsections: generic and building functions as well as specific functions to interface with FORM.

Generic functions

- **void setDim(int** `n`**)**

  Sets the group dimension.

- **int getDim()**

  Gets the group dimension.

- **void CleanGlobalDecl()**

  Cleans all tables with indices and function declarations.

- **void setVerbosity (Verbosity** `verb`**)**

  Sets verbosity level; verbosity options: SILENT, SUMMARIZE, VERBOSE, DEBUG_VERBOSE.

- **Verbosity getVerbosity()**

  Returns current verbosity level.

Building functions

- **DList b(**`i`**)/DList bb(**`i`**)**

  Declares a operator b_i; the index in **bb()** must be enclosed in quotation marks or passed as a std::string type.

- **DList bt(**`i`**)/DList bbt(**`i`**)**

  Declares a operator $b_i^\dagger$; the index in **bbt()** must be enclosed in quotation marks or passed as a std::string type.

- **DList delta(**`i,j`**)**

  Declares the δ_{ij} function.

- **DList identity**

  Declares the identity matrix.

- **Braket bra(**`A, B, C`**)**
  Braket ket(`A, B, C`**)**
  Braket braket(`A, B, C`**)**
  Braket free(`A, B, C`**)**

  The element A corresponds to the global index, B to the constant part (e.g. fields) and C to the operators b and $b^\dagger$, δ or the identity. The first entry is the sum of the number of all upper indices (positive counting) and lower indices (negative counting) present in the fields defined in

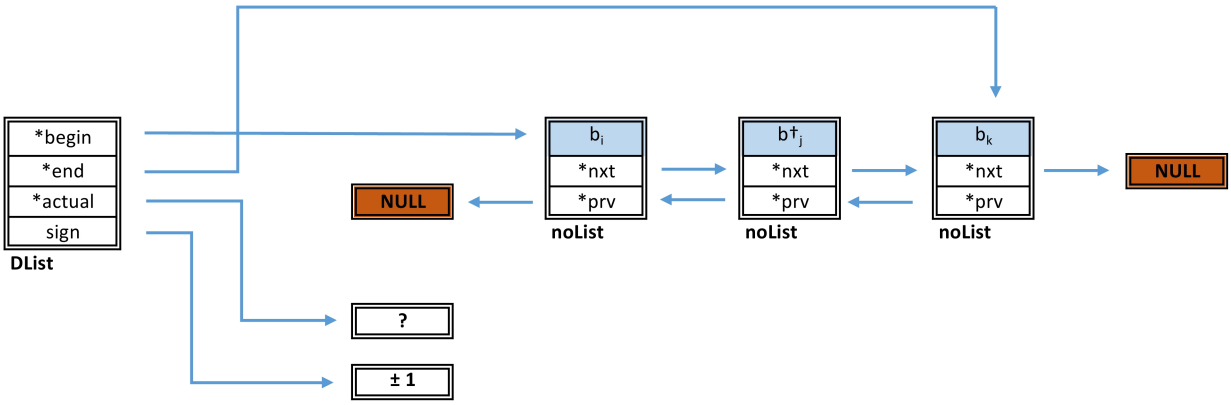

Figure 3: Representation of the bracket $\langle 0|b_i b^\dagger_j b_k|0\rangle$ in a linked data structure.

the function. This entry can be set to zero and if so we must call first the function **unsetSimplifyIndexSum**().

- **void evaluate**(**bool** onlydeltas=**true**)

  Evaluates expression, if onlydeltas is true then the expression is evaluated to deltas, if false the expression is evaluated to Levi-Civita tensors with eventual δ's.

- **Braket Bop**(std::**string** startid="i")

  Returns the operator B using generic indices.

  Braket BopIdnum()

  Returns the operator B using numeric indices.

- **void newId**(**string** i)

  Declares a new index.

- **void setSimplifyIndexSum**()
  void unsetSimplifyIndexSum()

  Activates/Deactivates internal simplifications based on the Braket Index sum. This option is activated by default.

Specific functions to interface with FORM

- std::**string Field**(A, B, C, D)

  This function is used to declare the field in FORM, where

  A field name;
  B number of upper indices;
  C number of lower indices;
  D field properties:
  - SYM: symmetric field without flavor index;
  - ASYM: antisymmetric field without flavor index;
  - SYM_WITH_FLAVOR: symmetric field with flavor index;
  - ASYM_WITH_FLAVOR: antisymmetric field with flavor index.

  Returns field name as it should be written in the constant **Braket** part.

  The convention to write a field in the constant part of a Braket is the following: for each field we assign a name, then we write the number of upper indices followed by the number of lower indices, then between parentheses we add the indices, the first index is always reserved for flavor if applicable, then we write the upper indices by the order they appear (left to right) followed by the lower indices (left to right). In the case we have some ambiguity concerning the symmetric or antisymmetric nature of the indices, we add the s letter for those that are symmetric just after the field name, e.g. the field M_{ijb} with symmetric i, j indices and flavor index b, must be written in the Braket constant part as Ms02(b,i,j) and declared to FORM as **Field**(M, 0, 2, SYM_WITH_FLAVOR).

- **void CallForm**(**Braket** &exp, **bool** print=**true**, **bool** all=**true**, **string** newidlabel="j")

  Creates the input file for FORM, run the FORM program and returns the result to an output file and/or to the screen.

- **void setFormRenumber**()

  Sets "renumber 1;" in FORM input file. This option is used to renumber indices in order to allow further simplifications. However, in large expressions this must be avoided since it increases the computational time in FORM. The best way to use it is simplify the expression with FORM with this option unset, and then send a second time to FORM with this option active. By default this option is unset.

  void unsetFormRenumber()

  Unsets "renumber 1;" in FORM input file.

- **void setFormIndexSum**()
  void unsetFormIndexSum()

Sets/Unsets the index sum in input FORM file. The set option is activated by default.

4. Work with SO2pin

In this section, we describe the installation of the SO2pin library and other tools that may be provided for the library to make further simplifications. We give in detail instructions how to use the SO2pin library in an standalone C++ program with a very simple example involving the group SO(4) just for illustration.

4.1. Download and installation

The SO2pin library project is hosted by Hepforge at `http://sospin.hepforge.org` under a GNU Lesser general public license.

The simplest way to compile the SO2pin library is:

1. `./configure --prefix=library_installation_path --with-form=FORM_path`

   The user can omit the FORM path declaration and set it after using **export** `PATH_TO_FORM=FORM_path` or put the binary FORM file in the folder where the user has its project.
2. `make`
3. `make install`
4. `make doxygen-doc` (optional - it generates SO2pin library documentation)

Inside the library folder there are several example files, in addition to the ones shown in this paper, to help with the use of the library. The FORM [49] binary files can be downloaded at `http://www.nikhef.nl/~form/`, after accepting the license agreement. They are available for LINUX (32-bits or 64-bits), Cygwin (32-bits) and Apple/Intel platforms. All the information concerning the installation is written in the README file. These procedures were successfully tested in LINUX and Mac OS X 10.10 (Yosemite).

4.2. Writing the first program

In this section we discuss how to write a first program example using the SO2pin library properly for SO(4). Although the group $\mathrm{SO}(4) \simeq \mathrm{SU}(2) \times \mathrm{SU}(2)$ is not particularly interesting for GUTs, it is invoked to illustrate the use of the library in a simpler way. Since SO(4) belongs to the family group SO($2N$) for N even, the two spinors in which the 4-dimensional space is broken are not related by conjugation. The general ket is given by,

$$|\psi\rangle = |0\rangle\, M \,+\, b_k^\dagger\,|0\rangle\, N^k \,+\, \frac{1}{2}\varepsilon^{ij} b_i^\dagger b_j^\dagger\,|0\rangle\,\overline{M}\,, \tag{58}$$

where $M, \overline{M} \sim 1$ and $N^i \sim 2$ in SU(2). So, the general ket in Eq. (58) can be decomposed as,

$$|\psi\rangle = |\psi_1\rangle + |\psi_2\rangle\ , \tag{59}$$

where $\Gamma_0\,|\psi_1\rangle = |\psi_1\rangle$ and $\Gamma_0\,|\psi_2\rangle = -\,|\psi_2\rangle$, which are written as

$$|\psi_1\rangle = |0\rangle\, M + \frac{1}{2}\varepsilon^{ij} b_i^\dagger b_j^\dagger\,|0\rangle\,\overline{M}\,, \tag{60}$$

$$|\psi_2\rangle = b_k^\dagger\,|0\rangle\, N^k\,. \tag{61}$$

Using the rules compiled in Section 2.3 for the transposition operations, we obtain the following expressions

$$\langle\psi_1^*| = M\,\langle 0| + \frac{1}{2}\varepsilon^{lm}\overline{M}\,\langle 0|\, b_m b_l\,, \tag{62}$$

$$\langle\psi_2^*| = N^n\,\langle 0|\, b_n\,. \tag{63}$$

As our first example program, we will address the calculation of $\left\langle\psi_1^*\middle|\, B\,\middle|\psi_1\right\rangle$. In order to better understand how the library works, let us start by showing first how to code $\left\langle\psi_1^*\middle|\psi_1\right\rangle$ ignoring the fields and taking into account only the creation and annihilation operators, i.e.,

$$\Big(\,\langle 0| + \langle 0|\, b_m b_l\Big)\Big(\,|0\rangle + b_i^\dagger b_j^\dagger\,|0\rangle\,\Big)\,. \tag{64}$$

To properly write the code for this expression, we first need to add the header file for the SO2pin library and the sospin **namespace**. After creating the main function we must define the group dimension in the beginning, **setDim**`(4)`, and clean up all the memory allocated using **CleanGlobalDecl**`()` before exiting from the program. The code to solve the problem in Eq. (64) is the following

```
#include <sospin/son.h>
using namespace sospin;

int main(int argc, char *argv[]){
 setDim(4);
 Braket left = identity;
 left += b(m)*b(l);
 Braket right = identity;
 right += bt(i)*bt(j);
 Braket res = left * right;
 res.evaluate();

 res.setON();
 std::cout<<"Result:\n"<< res <<std::endl;
 res.setOFF();

 CleanGlobalDecl();

 exit(0);
}
```

The **evaluate**`()` function can be used with or without the arguments: true or false. The **evaluate**`(`**false**`)` sets on the results written in terms of the Levi-Civita and it is only used in operations with **braket** type; **evaluate**`()`/**evaluate**`(`**true**`)` does the evaluation to delta functions and it can be used in **bra**`()`/**ket**`()`/**braket**`()` and `none` types. If we use more than one evaluation process, we need to maintain the type of evaluation chosen.

The functions **setON**`()` and **setOFF**`()` make possible the writing of `Local R?=` for each term in **Braket** expressions, in addition, each term of the expression is numbered. Note that in

the code written above, due to the way how it was declared, the terms need to be joined by using the += operation.

In order to compile and run the program, the user must pass to the C++ compiler the path to the SO2pin library and include folder, as for example (assuming the GCC compiler),

```
g++ -O3 -I/Sospin_PATH/include -L/Sospin_PATH
   /lib example.cpp -o example -lsospin
```

Running the program above, we will get the following result[3],

```
Local R1 = +1;
Local R2 = + bt(i) * bt(j);
Local R3 = + b(m) * b(l);
Local R4 = + d_(m,j) * d_(l,i)
         - d_(m,i) * d_(l,j)
         - d_(l,i) * bt(j) * b(m)
         + b(m) * bt(i) * bt(j) * b(l)
         + d_(l,j) * bt(i) * b(m);
```

This result is not in agreement with the rules given in Section 3. Following those rules and looking at the term above we see that all the terms containing operators must vanish. The reason why these terms appear in the result above is because we never declared the type of `left`, `right` and `res`, hence by default all these expressions are of type **free** (operation `none`). In order to properly solve Eq. (64) one needs to setup explicitly the type of each expression; we can declare them as

```
left.Type() = bra;
right.Type() = ket;
```

There is no need to declare the variable `res` because the operation `left*right` will automatically setup its type based on the product, i.e., the resulting type of `res` is **braket**.

A simpler and more complete way to declare the expressions in Eq. (64) is the following,

```
Braket left = Braket(identity, bra);
left += Braket(b(m)*b(l), bra);
Braket right = Braket(identity, ket);
right += Braket(bt(i)*bt(j), ket);
```

where we use the operation += for each contribution in different lines or

```
Braket left = Braket(identity, bra) + Braket(
   b(m)*b(l), bra);
Braket right = Braket(identity, ket) + Braket
   (bt(i)*bt(j), ket);
```

where we use the + operation for terms placed in the same line.

Running the program with the types properly setup, we get the expected result:

```
Local R1 = + 1;
Local R2 = + d_(m,j) * d_(l,i)
         - d_(m,i) * d_(l,j);
```

If we wish to evaluate the expression in Eq. (64) in such a way that the final result appears written in terms of the Levi-Civita tensors, we need simply set the argument of the **evaluate** function to false as `res.evaluate(false)`. The result will be given by the output

```
Local R1 = +1;
Local R2 = +e_(m,l)*e_(j,i);
```

As one can see, the final result written in terms of the Levi-Civita tensor is equal to the one written in terms of deltas but in a much more compact form. So, hereinafter we will just present the results written in terms of Levi-Civita tensors even though both methods are available.

Let us now discuss how to include in Eq. (64) the operator B of Eq. (A.19), i.e.,

$$\left(\langle 0| + \langle 0| b_m b_l\right) B \left(|0\rangle + b_i^\dagger b_j^\dagger |0\rangle\right). \tag{65}$$

Writing this code is rather simple because the SO2pin library already have a function to compute the operator B for any group SO($2N$), **Bop**(), therefore we only need to add this function to the code as:

```
Braket res = left * Bop() * right;
```

There is no need to set the group dimension in **Bop**() function because it is already done through the **setDim**() in the beginning of the code. If one wants to define the operator B by oneself, without using the predefined function, we can do it just by using the **free**() function and the rules given above. The result will be

```
Local R1 = 1/2*e_(i1,i2)*( -e_(i1,i2)*e_(j,i)
   );
Local R2 = 1/2*e_(i1,i2)*( +e_(i1,t1)*e_(i2,t1
   ) );
Local R3 = 1/2*e_(i1,i2)*(
          +d_(i1,i2)*e_(m,l)*e_(j,i)
          -d_(i1,i)*e_(m,l)*e_(j,i2)
          +d_(i1,j)*e_(m,l)*e_(i,i2)
);
Local R4 = 1/2*e_(i1,i2)*(
          +d_(i2,i)*e_(m,l)*e_(j,i1)
          -d_(i2,j)*e_(m,l)*e_(i,i1)
);
Local R5 = 1/2*e_(i1,i2)*( -e_(m,l)*e_(i2,i1)
   );
```

These results are quite large and clumsy so in order to simplify them we decided to include FORM as a final step. The inclusion of FORM is purely aesthetics and does not affect the computation procedures. In the case we consider it the last result become

```
Local R1 = +e_(i,j);
Local R2 = -e_(l,m);
```

After this introduction, we are ready to compute $\left\langle \psi_{1a}^* \middle| B \middle| \psi_{1b} \right\rangle$, i.e.,

$$\left(M_a \langle 0| + \frac{1}{2}\varepsilon^{lm}\overline{M}_a \langle 0| b_m b_l\right) B \left(|0\rangle M_b + \frac{1}{2}\varepsilon^{ij} b_i^\dagger b_j^\dagger |0\rangle \overline{M}_b\right), \tag{66}$$

where a and b are flavor indices.

[3] Note that, in order to save space, we altered slightly the aspect of the program's output.

In order to compute this example, we need to add the fields M and $\overline{M}$. This is done by using the functions **bra()**, **ket()**, **free()** and **braket()** defined in Section 3.2. To account for the changes due to the inclusion of the fields we need to substitute the codes given above by

```
Braket left = bra(0,M(a),identity);
left +=bra(0,Mb(a)*e_(1,m)/2,b(m)*b(1));
Braket right = ket(0,M(b),identity);
right +=ket(0,Mb(b)*e_(i,j)/2,bt(i)*bt(j));
```

where the field $\overline{M}$ is coded as `Mb`. The output result is

```
Local R1 = M(a)*1/2*e_(i1,i2)*Mb(b)*e_(i,j)
    /2*(-e_(i1,i2)*e_(j,i));
Local R2 = M(a)*1/2*e_(i1,i2)*M(b)*( +e_(i1,t1
    )*e_(i2,t1) );
Local R3 = Mb(a)*e_(1,m)/2*1/2*e_(i1,i2)*Mb(b)
    *e_(i,j)/2*(
        +d_(i1,i2)*e_(m,1)*e_(j,i)
        -d_(i1,i)*e_(m,1)*e_(j,i2)
        +d_(i1,j)*e_(m,1)*e_(i,i2)
);
Local R4 = Mb(a)*e_(1,m)/2*1/2*e_(i1,i2)*Mb(b)
    *e_(i,j)/2*(
        +d_(i2,i)*e_(m,1)*e_(j,i1)
        -d_(i2,j)*e_(m,1)*e_(i,i1)
);
Local R5 = Mb(a)*e_(1,m)/2*1/2*e_(i1,i2)*M(b)
    *( -e_(m,1)*e_(i2,i1) );
```

To make the final simplification we use the FORM program; before we call it to simplify our expression, we first need to declare explicitly all fields as well as all indices appearing only in the constant part, i.e., the indices a and b in this example. Therefore, we need to add the following code,

```
Field(M, 0, 0, ASYM_WITH_FLAVOR);
Field(Mb, 0, 0, ASYM_WITH_FLAVOR);
newId("a"); newId("b");
```

Once the fields M and $\overline{M}$ carry flavor, we need to set them as `ASYM_WITH_FLAVOR`, for more details see Section 3.2. To call the FORM program to simplify our expression we only need to write

```
CallForm(res,false, true, "j");
```

Note that the function `callForm()` sets **setOFF()** for the expression **Braket**. If the user have declared **setON()** previously, the user must set **setON()** again for that expression after `callForm()` . For a more detailed description about this function please see Section 3.2.

Running the program above we obtain the following result,

```
Local R1 = + M(j1) * Mb(j2);
Local R2 = - Mb(j1) * M(j2);
```

Note that j1 and j2 are not arguments but indices; they correspond to the flavor indices a and b. This is so because by default we consider that the indices are summed and hence they are renamed. If we want to avoid summed indices we must add the function **unsetFormIndexSum()** and then the result would be

```
Local R1 = + M(a) * Mb(b);
Local R2 = - Mb(a) * M(b);
```

For the sake of completeness, we give below the complete code to compute $\langle\psi^*_{1a}|B|\psi_{1b}\rangle$ in SO(4).

```
#include <sospin/son.h>
using namespace sospin;

int main(int argc, char *argv[]){

 setDim(4);

 Braket left = bra(0,M(a),identity);
 left +=bra(0,Mb(a)*e_(1,m)/2,b(m)*b(1));
 Braket right = ket(0,M(b),identity);
 right +=ket(0,Mb(b)*e_(i,j)/2,bt(i)*bt(j));

 Braket res = left * Bop() * right;
 res.evaluate(false);

 Field(M, 0, 0, ASYM_WITH_FLAVOR);
 Field(Mb, 0, 0, ASYM_WITH_FLAVOR);
 newId("a"); newId("b");

 unsetFormIndexSum();

 CallForm(res,false, true, "j");

 res.setON();
 std::cout<<"Result:\n"<< res <<std::endl;
 res.setOFF();

 CleanGlobalDecl();
 exit(0);
}
```

where the result is obviously $M_a\,\overline{M}_b - \overline{M}_a\,M_b$. In order to compute $\langle\psi^*_{1a}|B|\psi_{2b}\rangle$, $\langle\psi^*_2|B|\psi_1\rangle$ and $\langle\psi^*_{2a}|B|\psi_{2b}\rangle$ we just need to define $\langle\psi^*_{2a}|$ and $|\psi_{2b}\rangle$, and substitute it in the code above. Using Eq. (63) we define $\langle\psi^*_{2a}|$ as

```
Braket left = bra(1,N(a,n),b(n));
```

and using Eq. (61) we define $|\psi_{2b}\rangle$ as

```
Braket right = ket(1,N(b,k),bt(k));
```

The results are:

$$\begin{aligned}\langle\psi^*_{1a}|B|\psi_{2b}\rangle &= \langle\psi^*_2|B|\psi_1\rangle = 0\,,\\ \langle\psi^*_{2a}|B|\psi_{2b}\rangle &= \varepsilon^{ij}N^i_a N'^j_b\,.\end{aligned} \tag{67}$$

5. Examples

In this section we present two more complex examples to better explain how to use SO2qin library. We give one example in SO(4) with higher dimensional terms and one example in the context of SO(10) models.

Table 1: The SOgin code to compute $\langle \psi^*_{1a} | B\Gamma_\mu | \psi_{2b} \rangle \langle \psi^*_{1c} | B\Gamma_\mu | \psi_{2d} \rangle$ in SO(4) and the corresponding result. In this example we have used the operator B, **Bop()**, that uses generic indices, i and k.

```
#include <sospin/son.h>
using namespace sospin;

int main(int argc, char *argv[]){

 setDim(4);

 Braket L1, R1, L2, R2;
 Braket in1E, in1O, in2E, in2O, res;

 L1 = bra(0,M(a),identity);
 L1+= bra(0,Mb(a)*e_(i,j)/2,b(j)*b(i));
 R1 = ket(0,N10(b,k),bt(k));

 L2 = bra(0,M(c),identity);
 L2+= bra(0,Mb(c)*e_(l,m)/2,b(m)*b(l));
 R2 = ket(0,N10(d,o),bt(o));

 Field(M, 0, 0, ASYM_WITH_FLAVOR);
 Field(Mb, 0, 0, ASYM_WITH_FLAVOR);
 Field(N, 1, 0, ASYM_WITH_FLAVOR);

 newId("a"); newId("b");
 newId("c"); newId("d");

 in1E = L1 * Bop("i") * G(true, "j") * R1;
 in2E = L2 * Bop("k") * G(true, "j") * R2;
 in1O = L1 * Bop("i") * G(false,"j") * R1;
 in2O = L2 * Bop("k") * G(false,"j") * R2;

 in1E.evaluate();
 in2E.evaluate();
 in1O.evaluate();
 in2O.evaluate();

 res = in1E * in2E + in1O * in2O ;

 unsetFormIndexSum();
 CallForm(res,false, true, "i");

 res.setON();
 std::cout << "Output result:\n" << res << std::endl;

 CleanGlobalDecl();

 exit(0);
}
```

Output result:

```
Local R1 = +2*M(a)*N10(b,i)*Mb(c)*N10(d,j)*e_(i,j);
Local R2 = -2*Mb(a)*N10(b,i)*M(c)*N10(d,j)*e_(i,j);
```

5.1. SO(4)

Let us compute the following higher dimensional interaction term in SO(4):

$$\frac{Y_{ab}Y_{cd}}{\Lambda^2}\sum_{\mu=1}^{4}\left\langle\psi^*_{1a}\right|B\Gamma_\mu\left|\psi_{2b}\right\rangle\left\langle\psi^*_{1c}\right|B\Gamma_\mu\left|\psi_{2d}\right\rangle\,, \tag{68}$$

where Y_{ab}, Y_{cd} are Yukawa matrices, a, b, c, d are flavor indices and Λ is some high energy scale. The above equation can be expanded by rewriting the Γ_μ in terms of the operators b_i and $b_i^\dagger$ as given in Eq. (10)

$$\begin{aligned}\frac{Y_{ab}Y_{cd}}{\Lambda^2}\sum_{i=1}^{2}\Big(&\left\langle\psi^*_{1a}\right|B(b_i+b_i^\dagger)\left|\psi_{2b}\right\rangle\left\langle\psi^*_{1c}\right|B(b_i+b_i^\dagger)\left|\psi_{2d}\right\rangle\\ -&\left\langle\psi^*_{1a}\right|B(b_i-b_i^\dagger)\left|\psi_{2b}\right\rangle\left\langle\psi^*_{1c}\right|B(b_i-b_i^\dagger)\left|\psi_{2d}\right\rangle\Big).\end{aligned} \tag{69}$$

The code to compute Eq. (68) is given in Table 1 and the corresponding Γ_μ is given by taking into account the parity as

```
Braket G(bool even, string startid){
 if(even){
  Braket G_even = bb(startid +"1");
  G_even += bbt(startid +"1");
  return G_even;
 }
 Braket G_odd = bb(startid +"2");
 G_odd -= bbt(startid +"2");
 string constpart = "-i_";
 G_odd = G_odd * constpart;
 return G_odd;
}
```

Note that it is important to carefully define different indices among bracket expressions in order to avoid repeated indices, which could lead to a meaningless result. The result of the above code is then given as

$$2\frac{Y_{ab}Y_{cd}}{\Lambda^2}\left(M_aN_b^i\overline{M}_cN_d^j-\overline{M}_aN_b^iM_cN_d^j\right)\varepsilon_{ij}\,. \tag{70}$$

5.2. SO(10)

In this subsection, we give the example $16_a\,16_b\,120_H$ of SO(10) computed using the SO2qin library. In a first step we present the program by defining all quantities while in a second step we rewrite it using only the specific SO(10) functions already included in the library. The reducible 32 representation of SO(10) is given by

$$\begin{aligned}|\Psi\rangle=&|0\rangle\,\psi+b_i^\dagger|0\rangle\,\psi^i+\frac{1}{2}b_i^\dagger b_j^\dagger|0\rangle\,\psi^{ij}+\frac{1}{12}\varepsilon^{ijklm}b_k^\dagger b_l^\dagger b_m^\dagger|0\rangle\,\overline{\psi}_{ij}\\&+\frac{1}{24}\varepsilon^{ijklm}b_j^\dagger b_k^\dagger b_l^\dagger b_m^\dagger|0\rangle\,\overline{\psi}_i+b_1^\dagger b_2^\dagger b_3^\dagger b_4^\dagger b_5^\dagger|0\rangle\,\overline{\psi}\,.\end{aligned} \tag{71}$$

As already mentioned, in SO(10) the fermionic particles are usually assigned to the 16 dimensional representation which corresponds to the semi-spinor Ψ_+ while $\overline{16}\equiv\Psi_-$. The spinor representations are schematically given in Table 2 where $|\Psi_+\rangle$ and $|\Psi_-\rangle$ are given by

SO(10)	SU(5)	
$\|0\rangle$	1	M
$b_j^\dagger\|0\rangle$	5	M^i
$b_j^\dagger b_k^\dagger\|0\rangle$	10	M^{ij}
$b_j^\dagger b_k^\dagger b_l^\dagger\|0\rangle$	$\overline{10}$	$\overline{M}_{ij}$
$b_j^\dagger b_k^\dagger b_l^\dagger b_m^\dagger\|0\rangle$	$\overline{5}$	$\overline{M}_i$
$b_1^\dagger b_2^\dagger b_3^\dagger b_4^\dagger b_5^\dagger\|0\rangle$	1	$\overline{M}$
	dim = 2^5 = 32	

Table 2: The SO(10) states in terms of SU(5) fields.

$$|\Psi_+\rangle=|0\rangle\,M+\frac{1}{2}b_j^\dagger b_k^\dagger|0\rangle\,M^{jk}+\frac{1}{24}\varepsilon^{jklmn}b_k^\dagger b_l^\dagger b_m^\dagger b_n^\dagger|0\rangle\,\overline{M}_j\,, \tag{72}$$

and

$$|\Psi_-\rangle=b_i^\dagger|0\rangle\;M^i+\frac{1}{12}\varepsilon^{ijklm}\,b_k^\dagger b_l^\dagger b_m^\dagger|0\rangle\;\overline{M}_{ij}+b_1^\dagger b_2^\dagger b_3^\dagger b_4^\dagger b_5^\dagger|0\rangle\;\overline{M}\,, \tag{73}$$

while the transpose of $|\Psi\rangle$, represented as $\langle\Psi^*|$ are given by,

$$\left\langle\Psi^*_+\right|=M\,\langle0|+\frac{1}{2}M^{ij}\,\langle0|\,b_jb_i+\frac{1}{24}\varepsilon^{ijklm}\,\overline{M}_m\,\langle0|\,b_lb_kb_jb_i\,, \tag{74}$$

and

$$\left\langle\Psi^*_-\right|=M^i\,\langle0|\,b_i+\frac{1}{12}\varepsilon^{ijklm}\,\overline{M}_{ij}\,\langle0|\,b_mb_lb_k+\overline{M}\,\langle0|\,b_5b_4b_3b_2b_1\,, \tag{75}$$

where the flavor index was omitted.

The Yukawa term $Y_{ab}\;16_a\;16_b\;120_H$ is written as

$$\frac{1}{3!}Y_{ab}\left\langle\Psi^*_{+\,a}\right|B\,\Gamma_\mu\Gamma_\nu\Gamma_\rho\left|\Psi_{+\,b}\right\rangle\Phi_{\mu\nu\rho}\,. \tag{76}$$

In order to compute this expression using the SO2qin library, we write our main program as done before for SO(4) examples. The code to compute this example is given in Table 3. We start by including the `son.h` file, `sospin/son.h` in line 1, then we set the group dimension with the function **setDim**`(10)` (line 5). The expression for $|\Psi_+\rangle$ is given in Eq. (72) and is declared in line 11; $\langle\Psi^*_+|$ is given in Eq. (74) and declared in line 7 while the flavor indices a and b are declared in line 15 in Table 3.

Making use of the basic theorem [42] pointed out in Eq. (43), one can write the action of $\Gamma_\mu\Gamma_\nu\Gamma_\rho$ over the 120-dimensional Higgs field $\Phi_{\mu\nu\rho}$ which is given in Eq. (C.8) of Appendix C and defined in line 20 in Table 3. The charge conjugation operator, B, is in this code taken as the internal function **Bop**`()` (defined

Table 3: The SOSpin code to compute $16_a\,16_b\,120_H$, i.e., $\frac{1}{3!}\langle\Psi^*_{+a}|\,B\,\Gamma_\mu\Gamma_\nu\Gamma_\rho\,|\Psi_{+b}\rangle\,\Phi_{\mu\nu\rho}$ in SO(10) and the correspondig result.

```
#include <sospin/son.h>
using namespace sospin;

int main(int argc, char *argv[]){
 setDim(10);

 Braket psipbra = bra(0, M(a), identity);
 psipbra += bra(2, 1/2*M20(a,o,p),-b(o)*b(p));
 psipbra += bra(4,1/24*e_(o,p,q,r,s)*Mb01(a,o),b(p)*b(q)*b(r)*b(s));

 Braket psipket = ket(0,M(b), identity);
 psipket += ket(2,1/2 * M20(b,j,k), bt(j) * bt(k));
 psipket += ket(4, 1/24*e_(j,k,l,m,n)*Mb01(b,j), bt(k)*bt(l)*bt(m)*bt(n));

 newId("a");   newId("b");
 Field(M, 0, 0, ASYM_WITH_FLAVOR);
 Field(M, 2, 0, ASYM_WITH_FLAVOR);
 Field(Mb, 0, 1, ASYM_WITH_FLAVOR);

 Braket gamma = free(-3,1/6*(e_(r1,r2,r3,r4,r5)*H20(r4,r5)/sqrt(3)),b(r1)*b(r2)*b(r3));
 gamma += free(3,1/6*(e_(r1,r2,r3,r4,r5)*H02(r4,r5)/sqrt(3)),bt(r1)*bt(r2)*bt(r3));
 gamma += free(-1,(2*H12(r1,r2,r3)+d_(r1,r2)*H01(r3)-d_(r1,r3)*H01(r2))/(2*sqrt(3)), bt(r1)*b(r2
    )*b(r3));
 gamma += free(1,(2*H21(r1,r2,r3)+d_(r1,r3)*H10(r2)-d_(r2,r3)*H10(r1))/(2*sqrt(3)), bt(r1)*bt(r2
    )*b(r3));
 gamma += free(-1,2*H01(r1)/sqrt(3),-b(r1));
 gamma += free(1,2*H10(r1)/sqrt(3), bt(r1));
 newId("r4");   newId("r5");

 Field(H, 1, 0, ASYM);   Field(H, 0, 1, ASYM);
 Field(H, 0, 2, ASYM);   Field(H, 2, 0, ASYM);
 Field(H, 2, 1, ASYM);   Field(H, 1, 2, ASYM);

 Braket exp = psipbra * Bop() * gamma * psipket;
 exp.evaluate();
 CallForm(exp,true, true);

 CleanGlobalDecl();
 exit(0);
}
```

Output result:

```
R = + 2*M(j1)*H10(j2)*Mb01(j3,j2)*sqrt(1/3)*i_
  - M(j1)*H02(j2,j3)*M20(j4,j2,j3)*sqrt(1/3)*i_
  + M20(j1,j2,j3)*H01(j3)*Mb01(j4,j2)*sqrt(1/3)*i_
  + M20(j1,j2,j3)*H02(j2,j3)*M(j4)*sqrt(1/3)*i_
  + 1/4*M20(j1,j2,j3)*H21(j4,j5,j6)*M20(j7,j6,j8)*sqrt(1/3)*e_(j2,j3,j4,j5,j8)*i_
  - 1/4*M20(j1,j2,j3)*H21(j4,j5,j6)*M20(j7,j8,j6)*sqrt(1/3)*e_(j2,j3,j4,j5,j8)*i_
  - M20(j1,j2,j3)*H12(j4,j2,j3)*Mb01(j5,j4)*sqrt(1/3)*i_
  - 2*Mb01(j1,j2)*H10(j2)*M(j3)*sqrt(1/3)*i_
  - 1/2*Mb01(j1,j2)*H01(j3)*M20(j4,j2,j3)*sqrt(1/3)*i_
  + 1/2*Mb01(j1,j2)*H01(j3)*M20(j4,j3,j2)*sqrt(1/3)*i_
  + 2*Mb01(j1,j2)*H20(j2,j3)*Mb01(j4,j3)*sqrt(1/3)*i_
  + Mb01(j1,j2)*H12(j2,j3,j4)*M20(j5,j3,j4)*sqrt(1/3)*i_;
```

in Section 3.2). In line 26 we declare the indices that appear in `gamma` but are not defined automatically (i.e., the ones that appear only in fields, r4 and r5 in this case). The definition of fields in FORM language is done in lines 16 and 28. In line 32 the expression is evaluated, in line 34 we call the FORM to perform some final simplifications and in line 36 we clean temporary allocated memory with **CleanGlobalDecl()**.

Note that Y_{ab} carries flavor dependency, but it is not written explicitly in the code, and hence the final result presented in Table 3 can be simplified by performing the symmetrisation of the fields using

$$A_i\,B_j = \frac{1}{2}(A_i\,B_j + A_j\,B_i) + \frac{1}{2}(A_i\,B_j - A_j\,B_i). \tag{77}$$

If we work out the final result of Table 3 by using the expression above, we obtain

$$\begin{aligned} i\,\frac{2}{\sqrt{3}}\,Y^{-}_{ab}\,\Big(&2\,M_a\overline{M}_b\,H^i \;+\; M^{ij}_a\,M_b\,H_{ij} \\ &+\; \overline{M}_{ia}\,\overline{M}_{jb}\,H^{ij} \;-\; M^{ij}_a\,M_b\,H_j \\ &+\overline{M}_{ia}\,M^{jk}_b\,H^i_{jk} \;-\; \frac{1}{4}\,\varepsilon_{ijklm}\,M^{ij}_a\,M^{mn}_b\,H^{kl}_n\Big)\,, \end{aligned} \tag{78}$$

where $Y^{-}_{ab} \equiv \dfrac{Y_{ab} - Y_{ba}}{2}$.

Now we write down the specific function of SO(10) that are implemented in SO2qin library; these functions are defined in `tools/so10.h` file.

- **Braket** `GammaH(int n)`

  Gives the action of the `n` Γ-matrices acting over the Higgs field, as summarised in Appendix C; `n` is the number of Γ-matrices (runs from 0 to 5).

- **Braket** `psi_16p(OPMode mode, string id)`
  Braket `psi_16p(OPMode mode)`

  These functions give $\langle\psi^*_+|$, defined in Eq. (74), if the mode is bra and $|\psi_+\rangle$, defined in Eq. (72), if we select the ket mode.

- **Braket** `psi_16m(OPMode mode, string id)`
  Braket `psi_16m(OPMode mode)`

  These functions give $\langle\psi^*_-|$, defined in Eq. (75), if the mode is bra and $|\psi_-\rangle$, defined in Eq. (73) if the selected mode is ket.

- **Braket** `psi_144p(OPMode mode)`
  Braket `psi_144m(OPMode mode)`

  Functions for 144_+ and 144_-, defined for modes bra and ket. The complete expressions are given in Ref. [43].

In what follows, we rewrite the code presented in Table 3 just making use of the included functions described above.

```
#include <sospin/son.h>
#include <sospin/tools/so10.h>
using namespace sospin;

int main(int argc, char *argv[]){

  setDim(10);

  Braket res = psi_16p(bra) * Bop() *
               GammaH(3) * psi_16p(ket);

  res.evaluate();
  CallForm(res, true, true);

  CleanGlobalDecl();
  exit(0);
}
```

6. Conclusions

In this paper we have presented the SO2qin library, provided under the terms of the GNU Lesser General Public License as published by the Free Software Foundation. This is a C++ tool whose main goal is to decompose Yukawa interactions, invariant under SO($2N$), in terms of SU(N) fields. The library project is hosted by the Hepforge website `http://sospin.hepforge.org`.

This library relies on the oscillator expansion formalism that consists in expressing the SO(2N) spinor representations in terms of creation operators, $b^\dagger_i$, of a Grassmann algebra, acting on a vacuum state-vector. The SO2qin code simulates the non-commutativity of the operators and their products via the implementation of doubly-linked-list data structures. Such type of structures are the ideal method to deal with the usage of long chains of products of operators $b^\dagger_i$ and b_i. In this type of implementation, the sequences are linked through *nodes* that contains information about the previous and the next nodes of the sequence; this connections are called *links*. Moreover, the data storage in the memory does not need to be adjacent, this is one of the reason why the doubly-linked-lists led to high performances in our tests.

In order to understand the manipulations that SO2qin need to perform, we reviewed in detail the oscillator expansion formalism. Then, we applied the method for decomposing the SO($2N$) Yukawa terms with respect to SU(N) interactions. The general structure of SO2qin library was presented by listing the generic and devoted SO(10) functions. After explaining the installation and the writing of the first program, we showed in detail the usage of SO2qin through complete examples in both SO(4) and SO(10) frameworks. Additionally, we provided an higher dimensional field-operator example computed in the context of SO(4) to illustrate how such a term can be processed with this library. Finally, we described the functions available in SO2qin that were made to simplify the writing of spinors and their interactions specifically for SO(10) models. The code includes also functions to deal directly with the 144 and $\overline{144}$ representations of SO(10) and one can compute other quantities beyond the results already computed in Ref. [43].

We are planning to enhance the use of the memory in our library by implementing new forms of simplifications. We intend

to make the simplifications of the final expressions independent of external programs in order to reach more performance. Our future plans also include the extension of the library with specific functions, which are now only implemented for SO(10), automatised for a generic SO($2N$). Furthermore, although the SO2pin library was projected to cover the groups SO($2N$), it is easily adapted to groups SO($2N+1$) or other algebraic systems that are described by creation and annihilation operators.

Acknowledgments

D.E.C. would like to thank Palash B. Pal for enlightening discussions. C.S. would like to thank Jean-René Cudell for useful comments. D.E.C. and C.S. thank the Theory Unit of CERN Physics Department for the hospitality. D.E.C. also thanks Theory Unit of CERN for financial support. The work of D.E.C. is supported by Associação do Instituto Superior Técnico para a Investigação e Desenvolvimento (IST-ID) and Fundação para a Ciência e a Tecnologia (FCT) under the projects PTDC/FIS-NUC/0548/2012 and UID/FIS/00777/2013. The work of C.S. is supported by the Université de Liège and the EU in the context of the MSCA-COFUND-BeIPD project.

Appendix A. Clifford algebrae and SO($2N$)

In what follows we cover the Clifford algebra discussion given in Ref. [50] and we adapt it for the specific algebra obtained from SO($2N$). In general, we define the Clifford algebra as

$$\left\{\Gamma_\mu, \Gamma_\nu\right\} = 2\,\eta_\mu \delta_{\mu\nu}\,, \tag{A.1}$$

where η_μ is a real constant with $|\eta_\mu| = 1$. From Eq. (A.1) one gets $\Gamma_\mu^2 = \eta_\mu \mathbf{1}$ and

$$\Gamma_\mu^\dagger = \eta_\mu\, \Gamma_\mu\,. \tag{A.2}$$

For the spinor representation of SO($2N$), the corresponding Clifford algebra has $\eta_\mu = 1$ for any index μ, i.e.,

$$\left\{\Gamma_\mu, \Gamma_\nu\right\} = 2\,\delta_{\mu\nu}\,. \tag{A.3}$$

Moreover, one may construct a new independent matrix Γ_0, defined as

$$\Gamma_0 \equiv i^N \Gamma_1 \Gamma_2 \cdots \Gamma_{2N}\,, \tag{A.4}$$

which anticommutes with all Γ-matrices. This definition implies that $\Gamma_0^\dagger = \Gamma_0$, since

$$\begin{aligned}\Gamma_0^\dagger &= (-i)^N {\Gamma_{2N}}^\dagger \cdots {\Gamma_1}^\dagger = i^N (-1)^N \Gamma_{2N} \cdots \Gamma_1 \\ &= i^N (-1)^{N(2N-1)+N} \Gamma_1 \cdots \Gamma_{2N} = \Gamma_0\,,\end{aligned} \tag{A.5}$$

where one has used the following relation for non-repeating p Γ-matrices with $p \leq 2N$:

$$\Gamma_{m_p} \cdots \Gamma_{m_2} \Gamma_{m_1} = (-1)^{\frac{p(p-1)}{2}} \Gamma_{m_1} \Gamma_{m_2} \cdots \Gamma_{m_p}\,. \tag{A.6}$$

The exponent is obtained through the general formula for the sum of p elements of an arithmetic progression $u_1, \ldots, u_p$, i.e., $S_p = p\,\frac{u_1+u_p}{2}$. The first matrix Γ_{m_p} of the right-handed side of Eq. (A.6) has to anticommute $p-1$ times, the process ends with Γ_1, which does not need to move. One may then define the following projectors in analogy to the chiral projectors for fermion fields:

$$P_L \equiv \frac{1-\Gamma_0}{2}\,, \quad P_R \equiv \frac{1+\Gamma_0}{2}\,. \tag{A.7}$$

They are indeed projectors, since

$$\begin{aligned}&P_L + P_R = \mathbf{1}\,, \quad P_L^2 = P_L\,, \quad P_R^2 = P_R\,, \\ &P_L P_R = P_R P_L = 0\,.\end{aligned} \tag{A.8}$$

In order to deduce the main results of this appendix, construct the set C_D of all products of Γ-matrices of the Clifford algebra,

$$C_D = \{\pm 1, \pm\Gamma_{m_1}, \pm\Gamma_{m_1}\Gamma_{m_2}, \ldots, \pm\Gamma_0\}\,, \tag{A.9}$$

which is a finite group. Thus, we can enumerate all irreducible representations of the group C_D compatible with the algebra. In particular, the unidimensional representations of C_D cannot satisfy the anticommutation of Eq. (A.1). The order of the group C_D is given by

$$\mathrm{Ord}(C_D) = 2\sum_{i=0}^{2N} \binom{2N}{i} = 2^{2N+1}\,. \tag{A.10}$$

The finite cardinality of the group C_D ensures that there exists a basis where all matrices Γ_μ and their products can be taken unitary. From Eq. (A.2), one has in this basis $\Gamma_\mu^\dagger = \Gamma_\mu$ and $\Gamma_\mu^\intercal = \Gamma_\mu^*$.

For any finite group, the number of irreducible representations is equal to the number of classes. Thus, the enumeration of classes, n_c, of C_D is:

$$\begin{aligned}&\{1\}, \{-1\}, \{\pm\Gamma_1\}, \{\pm\Gamma_2\}, \cdots, \{\pm\Gamma_{2N}\}, \\ &\qquad \cdots, \{\pm\Gamma_1\Gamma_2\}, \cdots \{\pm\Gamma_0\}\,,\end{aligned} \tag{A.11}$$

which implies that

$$n_c = 1 + \sum_{i=0}^{2N} \binom{2N}{i} = 1 + 2^{2N}\,. \tag{A.12}$$

We conclude that the finite group C_D has $1 + 2^{2N}$ irreducible and nonequivalent representations. In order to determine how many unidimensional representations of C_D exist, we recall that in any finite group, the number of unidimensional representations is given by the ratio of number of the elements of the group and the elements of its commutator subgroup. The commutator subgroup of C_D has two elements, namely $[C_D, C_D] = \{1, -1\}$. Thus, the number of unidimensional representations is $\#C_D/\#[C_D, C_D] = 2^{2N}$. We then obtain that there is only one non-unidimensional representation for the group

C_D. The non-trivial representation can be obtained through the relation

$$\underbrace{1^2 + 1^2 + \cdots + 1^2}_{2^{2N}} + n^2 \;=\; 2^{2N+1}\,, \tag{A.13}$$

which implies that $n = 2^N$. It is clear that all representations of the Clifford algebra are also representations of C_D and therefore the Clifford algebra has only one irreducible representation with dimension 2^N. It is interesting to verify that from a given irreducible representation of the Clifford algebra Γ_μ, the matrices Γ_μ^*, $\Gamma_\mu^\intercal$, and $-\Gamma_\mu^\intercal$ form also an irreducible representation of the algebra since

$$\left\{\Gamma_\mu^*, \Gamma_\nu^*\right\} \;=\; \left\{\Gamma_\mu^\intercal, \Gamma_\nu^\intercal\right\} \;=\; \left\{-\Gamma_\mu^\intercal, -\Gamma_\nu^\intercal\right\} \;=\; 2\,\delta_{\mu\nu}\,. \tag{A.14}$$

Due to the fact there is only one irreducible representation of the Clifford algebra, these three representations must be necessarily equivalents. Thus, there must exist matrices B and C such that

$$B^{-1}\,\Gamma_\mu^*\,B \;=\; \Gamma_\mu\,, \tag{A.15a}$$

$$B^{-1}\,\Gamma_\mu^\intercal\,B \;=\; \Gamma_\mu\,, \tag{A.15b}$$

$$C^{-1}\,\Gamma_\mu^\intercal\,C \;=\; -\Gamma_\mu\,. \tag{A.15c}$$

Note that the operator B is the same for both Eqs. (A.15a) and (A.15b) since $\Gamma_\mu^\intercal = \Gamma_\mu^*$ and the operator C can be written as

$$C \;=\; B\Gamma_0\,, \tag{A.16}$$

apart from an overall complex factor. This derivation ensures the existence of the operator B, which was used in Eq. (26).

We end this appendix by showing that B and C can be taken unitary and they are either antisymmetric or symmetric matrices, depending of the value of N. Taking the fact that $(\Gamma_\mu^\intercal)^\intercal = \Gamma_\mu$ and using Eq. (A.15a) one concludes

$$\left(B^{-1}B^\intercal\right)^{-1}\,\Gamma_\mu\left(B^{-1}B^\intercal\right) = \Gamma_\mu\,, \tag{A.17}$$

which implies that $B^{-1}B^\intercal$ commutes with all elements of the spinor representation $U(\omega)$ and therefore by the first Schur lemma one has

$$B^{-1}B^\intercal = \epsilon\,\mathbf{1}\,. \tag{A.18}$$

Without the loss of generality, we can refactor the matrix B to have $|B| = 1$. This choice implies that $|\epsilon| = 1$ and

$$B^\intercal = \epsilon B\,. \tag{A.19}$$

The constant ϵ is in fact real, since transposing Eq. (A.18) one has

$$\epsilon \;=\; B\,(B^{-1})^\intercal \;=\; B\,(B^\intercal)^{-1} \;=\; \epsilon^*\,. \tag{A.20}$$

Thus, one has $\epsilon = \pm 1$, where the sign will depend only on N. Instead, if we use the relation $(\Gamma_\mu^*)^* = \Gamma_\mu$, we get

$$\left[B^*B,\,\Gamma_\mu\right] \;=\; 0\,. \tag{A.21}$$

The product B^*B then commutes with all elements of the spinor representation $U(\omega)$. The first Schur lemma obliges that

$$B^*B = \epsilon'\,\mathbf{1}\,, \tag{A.22}$$

with $\epsilon' = \pm 1$, since $|B| = 1$ and taking the conjugation of Eq. (A.22)

$$\epsilon'^* \;=\; BB^* \;=\; BB^*BB^{-1} \;=\; \epsilon'\,. \tag{A.23}$$

The relation between ϵ' and ϵ can be established by observing that $(\Gamma_\mu^\intercal)^* = (\Gamma_\mu^*)^\intercal$, which implies

$$\left[B^\dagger B,\,\Gamma_\mu\right] \;=\; 0\,, \tag{A.24}$$

The combination $B^\dagger B$ commutes with all elements of the spinor representation $U(\omega)$. The first Schur lemma leads to $B^\dagger B = \lambda\mathbf{1}$ with $\lambda = 1$. This can be obtained since $|B| = 1$ (implying $|\lambda| = 1$) and for any vector $|v\rangle$ with norm 1 one has

$$\lambda \;=\; \langle v|\,B^\dagger B\,|v\rangle \;=\; |B\,|v\rangle|^2 \;\geq 0\,. \tag{A.25}$$

We then obtain that B is unitary and as a consequence that C is also unitary. Moreover, using the unitarity of B,

$$\mathbf{1} \;=\; B^\intercal B^* \;=\; \epsilon\epsilon' BB^{-1}\,, \tag{A.26}$$

one concludes that $\epsilon' = \epsilon$.

One may raise the question how to relate the sign of ϵ with the dimension of $\mathsf{SO}(2N)$. We start by noting that

$$\Gamma_0^\intercal \;=\; B\,\Gamma_0\,B^{-1} \;=\; C\,\Gamma_0\,C^{-1}\,, \tag{A.27}$$

and one deduces, using Eq. (A.16), that

$$C^\intercal \;=\; \epsilon\,C\,. \tag{A.28}$$

If one takes into account Eq. (A.6), one is able to write the following relation

$$\left(C\,\Gamma_{\mu_1}\Gamma_{\mu_2}\cdots\Gamma_{\mu_p}\right)^\intercal \;=\; \epsilon\,(-1)^{\frac{p(p+1)}{2}}\,C\,\Gamma_{\mu_1}\Gamma_{\mu_2}\cdots\Gamma_{\mu_p}\,, \tag{A.29}$$

for $p \leq 2N$. The above relation implies that the matrices $C\,\Gamma_{\mu_1}\Gamma_{\mu_2}\cdots\Gamma_{\mu_p}$ are either symmetric or antisymmetric matrices. On the other hand, the set $\{C\,\Gamma_{\mu_1}\Gamma_{\mu_2}\cdots\Gamma_{\mu_p}\}$ forms a basis of the vector space of all $2^N \times 2^N$ complex matrices, since all order product of distinct Γ-matrices are 2^N linearly independent matrices. Thus, the number of independent antisymmetric matrices is given by

$$\sum_{p=0}^{2N}\frac{1}{2}\left[1 - \epsilon\,(-1)^{\frac{p(p+1)}{2}}\right]\binom{2N}{p}. \tag{A.30}$$

For any $2^N \times 2^N$ complex matrices, the number of independent antisymmetric matrices is just given by $2^N(2^N-1)/2$, therefore one has

$$\sum_{p=0}^{2N}\frac{1}{2}\left[1 - \epsilon\,(-1)^{\frac{p(p+1)}{2}}\right]\binom{2N}{p} \;=\; \frac{2^N(2^N-1)}{2}\,. \tag{A.31}$$

The above equation gives a closed relation between ϵ and N. After some algebraic simplifications, one gets

$$\epsilon \;=\; \frac{1}{2^{N+1}}\sum_{p=0}^{2N}(-1)^{\frac{(p-2)(p-1)}{2}}\binom{2N}{p}. \tag{A.32}$$

Taking into account the following relation

$$(-1)^{\frac{(p-2)(p-1)}{2}} \;=\; -\frac{1}{2}\Big[(1+i)\,i^p \;+\; (1-i)\,(-i)^p\Big], \tag{A.33}$$

that can be verified by mathematical induction, one derives

$$\epsilon \;=\; \sqrt{2}\cos\frac{\pi}{4}(2N+1)\,. \tag{A.34}$$

This equation has the period $N \to N+4$, and implies that $\epsilon = 1$ for $N = 3,\ 4 \pmod 4$, while $\epsilon = -1$ for $N = 1,\ 2 \pmod 4$. For instance, in the case of SO(10), one has $B^{\intercal} = -B$ and $C^{\intercal} = -C$.

Appendix B. Clifford vs. Grassmann algebrae

In this appendix, we show the existence of a one-to-one correspondence between Clifford and Grassmann algebrae, apart from an overall complex phases. In order to demonstrate this result, let us take an arbitrary Γ-matrix pair Γ_μ and Γ_ν of an abritrary Clifford algebra given in Eq. (A.1). Without the loss of generality, we shall take two consecutive gamma matrices,

$$\Big\{\Gamma_{2j-1},\Gamma_{2j}\Big\} = 0\,,\quad \Gamma_{2j-1}^2 = \eta_{2j-1}\mathbf{1}\,,\quad \Gamma_{2j}^2 = \eta_{2j}\mathbf{1}\,. \tag{B.1}$$

We identify then two possibilities $\eta_{2j-1} = \eta_{2j}$ and $\eta_{2j-1} = -\eta_{2j}$. It is then straightforward to verify that there exist a linear combination $b_j = \alpha\,\Gamma_{2j-1} + \beta\,\Gamma_{2j}$ such that they generate a Grassmann algebra,

$$\Big\{b_j, b_j^\dagger\Big\} = \mathbf{1}\,,\quad b_j^2 \;=\; 0\,. \tag{B.2}$$

Thus, when $\eta \equiv \eta_{2j-1} = \eta_{2j}$ one has

$$b_j \;=\; \frac{1}{2}\left(i\,\Gamma_{2j-1} \;+\; \Gamma_{2j}\right), \tag{B.3}$$

which then implies

$$b_j^\dagger \;=\; \frac{\eta}{2}\left(-i\,\Gamma_{2j-1} \;+\; \Gamma_{2j}\right), \tag{B.4}$$

where we have used the result that $\Gamma_a^\dagger \;=\; \eta_a\Gamma_a$. The two Eqs. (B.3) and (B.4) can be easily inverted so that the Γ_{2j-1} and Γ_{2j} are written in terms of b_j and $b_j^\dagger$ as

$$\Gamma_{2j-1} \;=\; -i(b_j \;-\; \eta\,b_j^\dagger)\,,\quad \Gamma_{2j} \;=\; b_j \;+\; \eta\,b_j^\dagger\,. \tag{B.5}$$

In the case of having $\eta_{2j-1} = -\,\eta_{2j}$, one has instead

$$b_j \;=\; \frac{1}{2}\left(\Gamma_{2j-1} \;+\; \Gamma_{2j}\right),\quad b_j^\dagger \;=\; \frac{\eta}{2}\left(\Gamma_{2j-1} \;-\; \Gamma_{2j}\right), \tag{B.6}$$

or the inverted system of equations

$$\Gamma_{2j-1} \;=\; b_j \;-\; \eta\,b_j^\dagger\,,\quad \Gamma_{2j} \;=\; b_j \;+\; \eta\,b_j^\dagger\,. \tag{B.7}$$

The fact that two distinct pair of Γ-matrices anticommutes, it guarantees that the operators b_j and $b_j^\dagger$, thus constructed, satisfy fully the Grasmann algebra, i.e.,

$$\Big\{b_j, b_k^\dagger\Big\} = \delta_{jk}\,\mathbf{1}\,,\quad \Big\{b_j, b_k\Big\} \;=\; \Big\{b_j^\dagger, b_k^\dagger\Big\} \;=\; 0\,. \tag{B.8}$$

We have shown that for any Clifford algebra one can write a set of creation and annihilation operators and therefore the SO2n library can be easily adapt for those cases.

Appendix C. SO(10) compendium

As a matter of completeness, in this section we compile some SO(10) details and functions not written in the previous chapters. In particular we summarise [41, 42] the action of Γ-matrices on the Higgs fields, ϕ, as well as the action of the operator B on $\langle\psi_+^*|$ and $\langle\psi_-^*|$.

The generic expression for the operator B is given in Eq. (28) and in what follows we list the action of the operator B on $\langle\psi_+^*|$ and $\langle\psi_-^*|$ given in Eqs. (72) to (75), respectively.

$$\begin{aligned}\langle\Psi_+^*\big|\,B = &-i\,\overline{\psi}_n\,\langle 0|\,b_n - \frac{i}{12}\varepsilon^{njklm}\psi_{nj}\,\langle 0|\,b_k b_l b_m\\ &- i\,\psi\,\langle 0|\,b_1 b_2 b_3 b_4 b_5\,,\end{aligned} \tag{C.1}$$

and

$$\langle\Psi_-^*\big|\,B = \frac{i}{24}\varepsilon^{njklm}\psi^n\,\langle 0|\,b_j b_k b_l b_m \;+\; \frac{i}{2}\overline{\psi}_{nj}\,\langle 0|\,b_n b_j + i\,\overline{\psi}\,\langle 0|\,. \tag{C.2}$$

One sees from the equations above that the action of B on the bras $\langle\psi_+^*|$ and $\langle\psi_-^*|$ gives expressions which are consistent with the SU(5) convention for upper and lower indices.

We compile below the action of Γ-matrices on Higgs representations of different dimensions, which follows from Eq. (43) and Refs. [41, 42]. For the 10-dim Higgs representation, ϕ_μ, one has,

$$\Gamma_\mu\phi_\mu = b_n^\dagger\,\phi_{c_n} + b_n\,\phi_{\bar{c}_n}, \tag{C.3}$$

with the following normalisation coming from the SU(5) notation

$$\phi_{c_n} \;=\; \sqrt{2}H^n,\quad \phi_{\bar{c}_n} \;=\; \sqrt{2}H_n. \tag{C.4}$$

For the 45-dim representation [42], $\phi_{\mu\nu}$, one has,

$$i\,\Sigma_{\mu\nu}\phi_{\mu\nu} = b_i b_j\phi_{\bar{c}_i\bar{c}_j} + b_i^\dagger b_j^\dagger\phi_{c_i c_j} + 2b_i^\dagger b_j\phi_{c_i\bar{c}_j} - \phi_{c_n\bar{c}_n}\,, \tag{C.5}$$

where

$$\begin{aligned}&\phi_{c_n\bar{c}_n} = h\,,\quad \phi_{c_i\bar{c}_j} = h_j^i + \tfrac{1}{5}\delta_j^i h\,,\\ &\phi_{c_i c_j} = h^{ij}\,,\quad \phi_{\bar{c}_i\bar{c}_j} = h_{ij}\,,\end{aligned} \tag{C.6}$$

with the normalisation

$$\begin{aligned}&h \;=\; \sqrt{10}\,H\,,\quad h^{ij} \;=\; \sqrt{2}\,H^{ij},\\ &h_{ij} \;=\; \sqrt{2}\,H_{ij}\,,\quad h_j^i \;=\; \sqrt{2}\,H_j^i\,.\end{aligned} \tag{C.7}$$

For the 120-dim Higgs field [42], $\phi_{\mu\nu\lambda}$, one has,

$$\begin{aligned}\Gamma_\mu\Gamma_\nu\Gamma_\lambda\phi_{\mu\nu\lambda} &= b_i b_j b_k \phi_{\bar{c}_i\bar{c}_j\bar{c}_k} + b_i^\dagger b_j^\dagger b_k^\dagger \phi_{c_i c_j c_k} \\ &+ 3(b_i^\dagger b_j b_k \phi_{c_i\bar{c}_j\bar{c}_k} + b_i^\dagger b_j^\dagger b_k \phi_{c_i c_j \bar{c}_k}) \\ &+ (3 b_i \phi_{\bar{c}_n c_n \bar{c}_i} + 3 b_i^\dagger \phi_{\bar{c}_n c_n \bar{c}_i}) \,,\end{aligned} \tag{C.8}$$

where

$$\phi_{c_i c_j \bar{c}_k} = h^{ij}_k + \frac{1}{4}\left(\delta^i_k h^j - \delta^j_k h^i\right) , \tag{C.9}$$

$$\phi_{c_i \bar{c}_j \bar{c}_k} = h^i_{jk} - \frac{1}{4}\left(\delta^i_j h_k - \delta^i_k h_j\right) , \tag{C.10}$$

$$\phi_{c_i c_j c_k} = \varepsilon^{ijklm} h_{lm} \,, \tag{C.11}$$

$$\phi_{\bar{c}_i \bar{c}_j \bar{c}_k} = \varepsilon_{ijklm} h^{lm} \,, \tag{C.12}$$

$$\phi_{\bar{c}_n c_n \bar{c}_i} = h^i \,, \tag{C.13}$$

$$\phi_{\bar{c}_n c_n \bar{c}_i} = -h_i \,, \tag{C.14}$$

with the following normalisation

$$\begin{aligned} h^i &= \tfrac{4}{\sqrt{3}} H^i \,, \quad h_i = \tfrac{4}{\sqrt{3}} H_i \,, \\ h^{ij} &= \tfrac{1}{\sqrt{3}} H^{ij} \,, \quad h_{ij} = \tfrac{1}{\sqrt{3}} H_{ij} \,, \\ h^{ij}_k &= \tfrac{2}{\sqrt{3}} H^{ij}_k \,, \quad h^i_{jk} = \tfrac{2}{\sqrt{3}} H^i_{jk} \,. \end{aligned} \tag{C.15}$$

Note that we have corrected the signs of the tensors in Eqs. (C.10) and (C.14).

For the 210-dim representation [42], $\phi_{\mu\nu\rho\lambda}$, one has,

$$\begin{aligned}\Gamma_\mu\Gamma_\nu\Gamma_\rho\Gamma_\lambda\,\phi_{\mu\nu\rho\lambda} =& 4\, b_i^\dagger b_j^\dagger b_k^\dagger b_l \phi_{c_i c_j c_k \bar{c}_l} + 4\, b_i^\dagger b_j b_k b_l \phi_{c_i \bar{c}_j \bar{c}_k \bar{c}_l} \\ &+ b_i^\dagger b_j^\dagger b_k^\dagger b_l^\dagger \phi_{c_i c_j c_k c_l} + b_i b_j b_k b_l \phi_{\bar{c}_i \bar{c}_j \bar{c}_k \bar{c}_l} \\ &- 6\, b_i^\dagger b_j^\dagger \phi_{c_i c_j c_m \bar{c}_m} + 6\, b_i b_j \phi_{\bar{c}_i \bar{c}_j c_m \bar{c}_m} \\ &+ 3\, \phi_{c_m \bar{c}_m c_n \bar{c}_n} - 12\, b_i^\dagger b_j \phi_{c_i \bar{c}_j c_m \bar{c}_m} \\ &+ 6\, b_i^\dagger b_j^\dagger b_k b_l \phi_{c_i c_j \bar{c}_k \bar{c}_l} \,,\end{aligned} \tag{C.16}$$

where

$$\begin{aligned} \phi_{c_m \bar{c}_m c_n \bar{c}_n} &= h, & \phi_{c_i \bar{c}_j c_m \bar{c}_m} &= h^i_j + \frac{1}{5}\delta^i_j h, \\ \phi_{\bar{c}_i \bar{c}_j \bar{c}_k \bar{c}_l} &= \frac{1}{24}\varepsilon_{ijklm} h^m, & \phi_{c_i c_j c_k c_l} &= \frac{1}{24}\varepsilon^{ijklm} h_m, \\ \phi_{c_i c_j c_m \bar{c}_m} &= h^{ij}, & \phi_{\bar{c}_i \bar{c}_j \bar{c}_m c_m} &= h_{ij}, \\ \phi_{c_i c_j \bar{c}_k \bar{c}_l} &= h^{ij}_{kl} + \frac{1}{3}\left(\delta^i_l h^j_k - \delta^i_k h^j_l + \delta^j_k h^i_l - \delta^j_l h^i_k\right) \\ &\quad + \frac{1}{20}\left(\delta^i_l \delta^j_k - \delta^i_k \delta^j_l\right) h, \\ \phi_{c_i c_j c_k \bar{c}_l} &= h^{ijk}_l + \frac{1}{3}\left(\delta^k_l h^{ij} - \delta^j_l h^{ik} + \delta^i_l h^{jk}\right), \\ \phi_{c_i \bar{c}_j \bar{c}_k \bar{c}_l} &= h^i_{jkl} + \frac{1}{3}\left(\delta^l_i h_{jk} - \delta^i_k h_{jl} + \delta^i_j h_{jk}\right) , \end{aligned} \tag{C.17}$$

with the normalisation

$$\begin{aligned} h &= \tfrac{4\sqrt{5}}{\sqrt{3}} H \,, & h^i &= 8\sqrt{6} H^i \,, & h_i &= 8\sqrt{6} H_i \,, \\ h^{ij} &= \sqrt{2} H^{ij} \,, & h_{ij} &= \sqrt{2} H_{ij} \,, & h^i_j &= \sqrt{2} H^i_j \,, \\ h^{ijk}_l &= \tfrac{\sqrt{2}}{\sqrt{3}} H^{ijk}_l \,, & h^i_{jkl} &= \tfrac{\sqrt{2}}{\sqrt{3}} H^i_{jkl} \,, & h^{ij}_{kl} &= \tfrac{\sqrt{2}}{\sqrt{3}} H^{ij}_{kl} \,. \end{aligned} \tag{C.18}$$

Finally, for the irreducible representations 126 ($\phi_{\mu\nu\lambda\rho\sigma}$) and $\overline{126}$ ($\bar{\phi}_{\mu\nu\lambda\rho\sigma}$), as it was observed in Section 2.2, one needs only to take into account the reducible 252 representation, since only one of its irreducible components survives [41],

$$\begin{aligned}\Gamma_\mu\Gamma_\nu\Gamma_\lambda\Gamma_\rho\Gamma_\sigma\,\Delta_{\mu\nu\lambda\rho\sigma} =& \varepsilon_{ijklm} b_i b_j b_k b_l b_m h \\ &+ \varepsilon^{ijklm}\, b_i^\dagger b_j^\dagger b_k^\dagger b_l^\dagger b_m^\dagger \bar{h} + 15\, b_i^\dagger h^i \\ &- 20\, b_i^\dagger b_n^\dagger b_n h^i + 5 b_i^\dagger b_n^\dagger b_n b_m^\dagger b_m h^i \\ &+ 15\, b_i h_i - 20\, b_n^\dagger b_n b_i h_i + 5\, b_n^\dagger b_n b_m^\dagger b_m b_i h_i) \\ &+ 10\,(b_i^\dagger b_j^\dagger b_k^\dagger h^{ijk} - b_i^\dagger b_j^\dagger b_k^\dagger b_n^\dagger b_n h^{ijk} \\ &+ b_i b_j b_k h_{ijk} - b_n^\dagger b_n b_i b_j b_k h_{ijk}) \\ &+ (60\, b_i^\dagger b_j^\dagger b_k h^{ij}_k - 30\, b_i^\dagger b_j^\dagger b_k b_n^\dagger b_n h^{ij}_k \\ &+ 60\, b_i^\dagger b_j b_k h^i_{jk} - 30 b_n^\dagger b_n b_i^\dagger b_j b_k h^i_{jk}) \\ &+ (5\, b_i^\dagger b_j^\dagger b_k^\dagger b_l^\dagger b_m h^{ijkl}_m + 5\, b_i^\dagger b_j b_k b_l b_m h^i_{jklm}) \\ &+ (10\, b_i^\dagger b_j^\dagger b_k^\dagger b_l b_m h^{ijk}_{lm} + 10\, b_i^\dagger b_j^\dagger b_k b_l b_m h^{ij}_{klm}) \,,\end{aligned} \tag{C.19}$$

where

$$\begin{aligned} h &= \frac{2}{\sqrt{15}} H \,, \quad h^i = \frac{4\sqrt{2}}{\sqrt{5}} H^i \,, \\ h^{ijk} &= \frac{\sqrt{2}}{\sqrt{15}} \varepsilon^{ijklm} H_{lm} \,, \quad h^i_{jklm} = \frac{\sqrt{2}}{\sqrt{15}} \varepsilon_{jklmn} H^{ni}_{(S)} \,, \\ h^i_{jk} &= \frac{2\sqrt{2}}{\sqrt{15}} H^i_{jk} \,, \quad f^{ijk}_{lm} = \frac{2}{\sqrt{15}} H^{ijk}_{lm} \,. \end{aligned} \tag{C.20}$$

The field $H^{ij}_{(S)}$ denotes the 15 representation of SU(5), which is a symmetric tensor.

Appendix D. Low-level implementations of the SO2qin library

In this section we list and describe all functions in the header files `dlist.h`, `index.h`, `braket.h`, `form.h` and `son.h`.

`dlist.h`

- **DList::DList(void)**
  DList::DList(const DList &L)
  DList::DList(int type, int i, int j)
  DList::DList(int type, int i)
  DList::~DList(void)

  The prototypes stand for the default, copy and specific constructors and the destructor, respectively, within the class **DList**. Note that the argument `type` defines the type of the node element. The arguments `i` and `j` are at the most two indices to complete the element. Pointers to the `noList`-type fields are initialised with `NULL`.

- **void DList::clear()**

  Deletes all nodes from **DList**.

- **void DList::negate()**

  Changes the sign of **DList**.

- **void DList::add**(elemType)

  Adds one node at the end of **DList** (scans the list) and updates the actual pointer to be the last node.

- **void DList::add_begin**(elemType)

  Adds one node in the beginning of **DList** and updates the actual pointer to be the new first node.

- **void DList::add_begin**(elemType)

  Adds one node in the beginning of **DList** and updates the actual pointer to be the new first node.

- **void DList::add_end**(elemType)

  Adds one node in the end of **DList** and updates the actual pointer to be the last node.

- **void DList::set**(elemType)

  Sets data (elemType) of the node being pointed by actual pointer.

- **void DList::set_begin**()

  Changes actual pointer to point at the first element of **DList** (beg pointer).

- **void DList::set_end**()

  Changes actual pointer to point at the last element of **DList** (end pointer).

- **void DList::set_sign**(**int** i)

  Sets the sign of **DList**.

- **void DList::join**(**DList**& L)

  Joins a **DList** to the end of the current **DList** (this) and updates the *actual* pointer to be the end of the final **DList**.

- **void DList::loop_right**()

  Shifts actual pointer to next node. If actual node is the end node, shift to beg node.

- **void DList::loop_left**()

  Shifts actual pointer to previous node. If actual node is the beg node, shift to end node.

- **DList DList::rearrange**()

  Creates and returns a new **DList** by copying nodes in **DList** ordered by type. The nodes that first appear in the new ordered **DList** are δ's (type=2) and then all other elements: b (type=0) and $b^\dagger$ (type=1) unordered. Constant elements are removed.

- **void DList::remove**(**unsigned int** type)

  Removes the first element with data.get_type()== type found in **DList**. Updates actual pointer to be the first node.

- **void DList::remove_actual**()

  Removes the element for which the actual pointer, *actual*, is pointing at in **DList**.

- **void DList::shift_right** ()

  Shifts actual pointer to next node. If actual node is the end node, stops.

- **void DList::shift_left** ()

  Shifts actual pointer to previous node. If actual node if first node (*begin*), stops.

- **void DList::swap_next**()

  Swaps the actual node with the next node of **DList**.

- elemType **DList::get**()

  Returns elemtype of the node being pointed by actual (current element).

- **int DList::getSign**()

  Returns the sign of **DList**.

- vector<**int**> **DList::getIds**()

  Creates and returns an integer vector sequence container with the ids (data fields) of b's and $b^\dagger$'s elements.

- **void DList::getBandBdaggerIds**(**bool** &BandBd, vector <**string**> &id0, vector<**string**> &id1, **int** &sign)

  Updates integer vector sequence containers id0 and id1 with ids (data fields) of b's and $b^\dagger$'s elements, respectively; the parameter sign is updated with the sign of **DList** and BandBd is a boolean which is true if **DList** contains at least one b or $b^\dagger$ and false otherwise.

- **void DList::getDeltaIds**(**bool** &AllDeltas, vector <**string**> &id0, vector<**string**> &id1, **int** &sign)

  Updates integer vector sequence containers id0 and id1 with the first and second ids (data fields) of δ elements, respectively; the parameter sign is updated with the sign of **DList** and AllDeltas is a boolean which is true if all elements in **DList** are of δ type and false otherwise.

- **void DList::getBandBdaggerAndDeltasIds**(vector <**string**> &id0, vector<**string**> &id1, vector< **string**> &id2, vector<**string**> &id3, **int** &sign)

  Updates integer vector sequence containers id0 and id1 with ids (data fields) of b's and $b^\dagger$'s elements, respectively; updates integer vector sequence containers id2 and id3 with first and second data fields of δ's elements, respectively. The parameter sign is updated with the sign of **DList**.

- **int DList::numDeltas**()

  Returns the number of elements of type δ (type=2).

- **int DList::numBs()**

  Returns the number of elements of type b (type=0).

- **bool DList::search_last (unsigned int type1)**

  Search the last element with data.get_type()==type1 found in **DList**. Returns true a node was found.

- **bool DList::search_first (unsigned int type1)**

  Search the first element with data.get_type()==type found in **DList** and returns true if the symbol is not the first element, and false otherwise.

- **bool DList::search_first (unsigned int type0, unsigned int type1)**

  Checks for the order of appearance in **DList** of types type0 and type1. Returns true if order of appearance is the same as the parameter's order and false otherwise.

- **bool DList::search_elem(unsigned int type1)**

  Searches for the element with data.get_type()==type found in **DList**. Returns true if the element is found, false otherwise.

- **bool DList::check()**

  Verifies if the number of b's and $b^\dagger$'s matches and whether the number each one is less or equal than N of SO($2N$).

- **bool DList::checkDeltaIndex()**

  Checks the indexes of δ elements and if the indexes in δ are equal. Returns true if each δ is not zero, false otherwise.

- **bool DList::check_num()**

  Verifies if the number of b's and $b^\dagger$'s is less or equal than N of SO($2N$) if so the function returns true if not returns false.

- **bool DList::check_same_num()**

  Verifies if the number of b's and $b^\dagger$'s matchs; returns true if they match and false otherwise.

- **bool DList::isActualLast()**

  Returns true if actual pointer is pointing to the last (*end*) node of **DList**.

- **bool DList::isEmpty()**

  Returns true if **DList** has no nodes.

- **bool DList::hasNoDeltas()**

  Returns true if there is no elements of type δ in **DList**.

- **bool DList::hasOnlyDeltas()**

  Returns true if all nodes in **DList** are of δ type.

- **bool DList::hasRepeatedIndex()**

  Returns true if there is elements with the same id (data fields) in the **DList** (repeated ids).

- **DList& DList::operator=(const DList& L)**

  Copies a **DList**.

- **const DList DList::operator-()const**

  Negates operator, change sign of **DList**.

- **friend DList* copy(DList *L)**

  Creates and returns a pointer to a new copy of a **DList**.

- **friend DList contract_deltas (DList &L, bool braketmode)**

  Applies the following identity $b_i * b_j^\dagger = \delta_{i,j} - b_j^\dagger * b_i$, input **DList**&L keeps delta term and the function returns the swapped term; the parameter braketmode- if true and if last element in **DList** is a b, then the L is cleared. Returns expression with $b_i * b_j^\dagger$ swapped or empty expression if b is the last term in L.

- **friend DList ordering(DList &L, bool braketmode)**

  Order only the b's (to the left hand side) and $b^\dagger$'s (to the right hand side) terms. Applies the following identity $b_j^\dagger * b_i = \delta_{i,j} - b_i * b_j^\dagger$, input **DList** L keeps delta term and function returns the swapped term. braketmode - if true and if last element in **DList** is a b, then L is cleared. Returns expression with $b_j^\dagger * b_i$ swapped or empty expression if b is the last term in L.

- **friend string printDeltas (DList &L)**

  Creates and returns a string with the deltas and constants of a **DList**.

- **friend DList& operator*(DList& L, elemType j)**

  Adds element j to the end of **DList** and returns a pointer to **DList**.

- **friend DList operator*(const DList& L, const DList &M)**

  Creates and returns a new **DList** that joins two DLists by order of parameters.

- **friend DList& operator-(DList& L, elemType j)**

  Negates the sign of **DList** and adds the elemtype j at end of it.

- **friend DList& operator,(DList& L, elemType j)**

  Adds element j to the end of **DList** and returns a pointer to **DList**.

- **friend ostream& operator<<(ostream& out, DList &L)**

  Sends to output stream ostream a string with the corresponding expression of the **DList**.

- **friend** `ostream&` **operator**<<(`ostream& out, DList *L`)

  Sends to output `stream ostream` a string with the corresponding expression of the **DList**.

- **friend** **DList**`&` **operator**<<(**DList** `&L, elemType j`)

  Adds element `j` to the end of **DList** and returns a pointer to **DList**.

- **friend** **DList**`&` **operator**<<(**DList** `&L,` **DList** `&M`)

  Copies **DList**; creates and returns a new **DList** with nodes of both old and new `DLists`. Sign is the product of both products.

- **friend** **bool** **operator**==(**DList** `&L,` **DList** `&M`)

  Returns true if two `DLists` are equal.

index.h.

- **int** **newIdx**(**int** `i`)

  See definition in Section 3.2.

- **int** **newIdx**(**string** `i`)

  Stores a new index of type **string**.

- **void** **newId**(**string** `i`)

  Stores a new index of type **string**.

- **string** **getIdx**(**int** `i`)

  Returns the index placed at the position `i`.

- **int** **Idx_size**()

  Returns index list size.

- **string** **IndexList**()

  Returns index list in string of the form "Indices ?,...,?".

- **void** **printIDX**()

  Prints index list.

- **string** **makeId**(**string** `a,` **int** `id`)

  Returns `a+id` in a string format.

- **template**<**class** `T>` **string** **ToString**(`T number`)

  Converts to string.

braket.*h*.

- **void** **setSimplifyIndexSum**()

  See definition in Section 3.2.

- **void** **unsetSimplifyIndexSum**()

  See definition in Section 3.2.

- **BraketOneTerm**::**BraketOneTerm**()

  Constructor.

- **BraketOneTerm**::**BraketOneTerm**(**const** **DList** `&d`)

  Constructor without constant part and index zero; `d` is a **DList** expression.

- **BraketOneTerm**::**BraketOneTerm**(**int** `indexin,` **string** `constpartin,` **const** **DList** `&d`)

  Constructor; `indexin` is the index of the expression, `constpartin` is the constant part and `d` is a **DList** expression.

- **BraketOneTerm**::**BraketOneTerm**(**int** `indexin,` **string** `constpartin, list`<**DList**> `termin`)

  Constructor; `indexin` is the index of the expression, `constpartin` is the constant part and `termin` is a `list`<**DList**> expression.

- **BraketOneTerm**::**BraketOneTerm**(**int** `indexin,` **string** `constpartin,` **BraketOneTerm**`& termin`)

  Constructor; `indexin` is the index of the expression, `constpartin` is the constant part and `termin` is a **BraketOneTerm** expression.

- **BraketOneTerm**::~**BraketOneTerm**()

  Destructor; clears all allocated memory.

- **void** **BraketOneTerm**::**clear**()

  Clears all allocated memory and sets default parameters.

- `list`<**DList**>`&` **BraketOneTerm**::**GetTerm**()

  Returns and sets the term part.

- **string**`&` **BraketOneTerm**::**GetConst**()

  Returns (and sets) the constant part.

- **int**`&` **BraketOneTerm**::**GetIndex**()

  Returns (and sets) the index sum part.

- **bool** **BraketOneTerm**::**Simplify**(`OPMode oper`)

  Simplifies current expression term.

  `oper` term mode (bra, braket, ket or none)

  return true if expression term is empty, false otherwise

- **bool** **BraketOneTerm**::**checkindex**()

  Checks global index in expression term; returns true if |index| is equal to 0 or N of SO($2N$), otherwise returns false.

- **void** **BraketOneTerm**::**expfromForm**(**string** `a`)

  To pass an expression from form.

- **void BraketOneTerm::rearrange()**

  Orders nodes of **DList** in **Braket**. First deltas and then b's and $b^\dagger$'s, and removes the identity node when δ, b or $b^\dagger$ are present.

- **bool BraketOneTerm::isempty()**

  Returns true if expression is `empty`.

- **bool BraketOneTerm::EvaluateToDeltas(OPMode oper)**

  Evaluates the expression to deltas; the mode of `oper` can be **bra**, **braket**, **ket** or `none`. This function returns true if the term is empty (or gives zero) otherwise returns false.

- **bool BraketOneTerm::EvaluateToLeviCivita(OPMode oper)**

  Evaluates the expression to Levi-Civita tensors with eventual δ's; the mode of `oper` can be **bra**, **braket**, **ket** or `none`. This function returns true if the term is empty (or gives zero) otherwise returns false.

- **void BraketOneTerm::neg()**

  Negates **BraketOneTerm**.

- **BraketOneTerm BraketOneTerm::operator*(const string constval)**

  Overloads **operator** for **BraketOneTerm** `* constval`.

- **BraketOneTerm BraketOneTerm::operator*=(const string constval)**

  Overloads **operator** for **BraketOneTerm** `*= constval`.

- **BraketOneTerm BraketOneTerm::operator*(const BraketOneTerm &L)**

  Overloads **operator** for **BraketOneTerm** `* L`.

- **BraketOneTerm BraketOneTerm::operator*=(const BraketOneTerm &L)**

  Overload **operator** for **BraketOneTerm** `*= L`.

- **friend BraketOneTerm operator-(const BraketOneTerm &L)**

  Negates operator.

- **friend ostream& operator<<(ostream& out, const BraketOneTerm &L)**

  Stream operator.

- **Braket::Braket(void)**

  Constructor; the default expression mode is none.

- **Braket::Braket(const DList &d0)**

  Constructor without constant part and index zero; `d0` is **DList** expression.

- **Braket::Braket(int id, string a, DList d0)**

  Constructor; the default expression mode is none and `id` is the index of the expression, `a` is the constant part and `d0` is the **DList** expression.

- **Braket::Braket(int id, string a, DList d0, OPMode op)**

  Constructor, default expression mode is none.

  `id` - index of the expression

  `a` - constant part

  `d0` - **DList** expression

  `op` - Braket type, i.e., bra/ket/braket/none

- **Braket::Braket(const Braket &L)**

  Constructor.

  `L` - **Braket** expression

- **Braket::Braket(int id, string a, const Braket &L, OPMode op)**

  Constructor.

  `id` - index of the expression

  `a` - constant part

  `L` - **Braket** expression, ignores current constant part of `L`

  `op` - **Braket** type, i.e., bra/ket/braket/none

- **Braket::Braket(BraketOneTerm term)**

  Constructor, default expression mode is none.

  `term` - **BraketOneTerm** expression

- **Braket::Braket(BraketOneTerm term, OPMode op)**

  Constructor.

  `term` - **BraketOneTerm** expression

  op - **Braket** type, i.e., bra/ket/braket/none

- **Braket::~Braket()**

  Destructor, clears all allocated memory.

- **void Braket::clear()**

  Clears all allocated memory and sets default parameters.

- **void Braket::expfromForm(vector<string> a)**

  To pass an expression from form.

- **OPMode& Braket::Type()**

  Returns the current expression type, it also allows to set new expression type. Expression types: bra/ket/braket or none.

- **void Braket::mode()**

  Prints the **Braket** expression mode, ie, the type of **Braket**: bra/ket/braket or none.

- **void Braket::evaluate(bool onlydeltas=true)**

  See Section 3.2.

- **void Braket::simplify()**

  Simplifies expression. Applies the following rules: $b_i|0\rangle = 0$ and $\langle 0|b_j^\dagger = 0$. In $\langle 0|...|0\rangle$ the number of b_i must be equal to the number of $b_j^\dagger$. It also checks for numeric deltas and evaluates them.

- **void Braket::rearrange()**

  Orders nodes of **DList** in Braket. First deltas and then b's and $b^\dagger$'s and removes the identity node when deltas, b or $b^\dagger$ are present.

- **void Braket::checkDeltaIndex()**

  Checks index in the deltas.

- **void Braket::gindexsetnull()**

  Sets to zero the index sum of each expression term.

- **void Braket::checkindex()**

  Checks global index in expression term if **setSimplifyIndexSum()** or **FlagSimplifyGlobalIndexSum()** is active, returns true if |index| is equal to 0 or N of SO($2N$), otherwise returns false.

- **void Braket::setON()**

  Activates expression term numbering for output writing for each term `Local R?=`.

- **void Braket::setOFF()**

  Deactivates expression term numbering for output writing for each term `Local R?=`.

- **int Braket::size()**

  Returns number of terms in current expression.

- **BraketOneTerm& Braket::Get(int pos)**

  Returns expression term at position given by `pos`.

- **int& Braket::GetIndex(int pos)**

  Returns/sets the index sum of the term given by `pos`.

- **Braket Braket::operator=(const Braket &L)**

  Overloads operator for **Braket** `= L`.

- **Braket Braket::operator+(const Braket &L)**

  Overloads operator for **Braket** `+ L`.

- **Braket Braket::operator+=(const Braket &L)**

  Overloads operator for **Braket** `+= L`.

- **Braket Braket::operator-(const Braket &L)**

  Overloads operator for **Braket** `- L`.

- **Braket Braket::operator-=(const Braket &L)**

  Overloads operator for **Braket** `-= L`.

- **Braket Braket::operator*(const Braket &L)**

  Overloads operator for **Braket** `* L`.

- **Braket& Braket::operator*=(const Braket &L)**

  Overloads operator for **Braket** `*= L`.

- **Braket Braket::operator*(const string constval)**

  Overloads operator for **Braket** `* constval`, i.e., the constant part.

- **Braket Braket::operator*=(const string constval)**

  Overloads operator for **Braket** `*= constval`, i.e., the constant part.

- **friend Braket operator-(const Braket &L)**

  Overloads operator for negate, `-L`.

- **friend OPMode operator*(const OPMode a, const OPMode b)**

  Calculates the mode for the multiplication. Returns mode of the multiplication, if this results in an invalid operation the program exit. `a` is the mode of the `left operand`, **b** is the mode of the `right operand`.

- **friend OPMode operator+(const OPMode a, const OPMode b)**

  Calculates the mode for the sum. Returns mode of the sum, if this results in an invalid operation the program exit. `a` is mode of the `left operand` e `b` is mode of the `right operand`.

- **friend OPMode operator-(const OPMode a, const OPMode b)**

  Calculates the mode for the subtraction. Returns mode of the subtraction, if this results in an invalid operation the program exit. `a` is the mode of the `left operand` e `b` is mode of the `right operand`.

- **friend ostream& operator<<(ostream& out, const Braket &L)**

  Writes expression to `ostream`.

- **friend string& operator<<(string& out, const Braket &L)**

  Writes expression to **string**.

- **friend string& operator+ (string& out, const Braket &L)**

  Writes expression to **string**.

- `ostream& operator<<(ostream& out, const OPMode &a)`

  Gets the mode of the expression `a` of the current expression and returns the mode in `ostream`.

form.h

- `void setFormRenumber()`
  `void unsetFormRenumber()`
  `void setFormIndexSum()`
  `void unsetFormIndexSum()`

  See Section 3.2.

- `string FormField(const string fieldname, const unsigned int numUpperIds, const unsigned int numLowerIds, const FuncProp funcp)`

  See Section 3.2.

- `void CallForm(Braket &exp, bool print=true, bool all=true, string newidlabel="j")`

  See Section 3.2.

- `ToForm::ToForm(void)`

  Constructor.

- `ToForm::~ToForm()`

  Destructor.

- `void ToForm::clear()`

  Clears all allocated memory and sets default values.

- `bool ToForm::function(string f)`

  Stores a field name.

- `void ToForm::contractions(string f)`

  Stores all field contractions.

- `ToForm::string getFC()`

  Returns all type of contractions for the fields.

- `string ToForm::getFunction()`

  Returns all the field names.

- `void ToForm::setFilename(string name)`

  Sets the beginning of a input/output FORM file.

- `string ToForm::file()`

  Returns the beginning of a input/output FORM file.

- `string& ToForm::rpath()`

  Returns the full path name and form binary file

- `void ToForm::run(Braket &exp, bool print, bool all, string newidlabel)`

  Simplify expression in FORM. Creates file input for FORM, runs the FORM program and returns the result to file and/or screen.

  `exp` - Braket expression to be simplified in FORM, the result is written back.

  `print` - if TRUE prints final result to screen

  `all` - if TRUE writes all the expression members separately in ouput FORM file, if FALSE only writes the full result together

  `newidlabel` - label to be used when the option to sum index is active

- `bool ToForm::getIndexSum()`

  Returns the state of the `indexSum` flag.

- `void ToForm::setIndexSum(bool flag)`

  Sets the state of the `indexSum` flag .

- `void ToForm::setRenumber(bool flag=true)`

  Sets "renumber 1;" in FORM input file. This option is used to renumber index in order to allow further simplifications. However, in big expressions this must be avoid since it increases the computational time in FORM. The best way to use is simplify the expression with FORM with this option unset, and then send a second time to FORM with this option active. By default this option is unset.

- `bool ToForm::getRenumberOption()`

  Returns the state of the `formRenumber` flag.

- `ToForm& ToForm::operator<<(const string &func)`

  Overloads operator for `ToForm << func`.

- `ToForm& ToForm::operator+(const string &func)`

  Overloads operator for `ToForm + func`.

son.h

- `void setDim(int n)`
  `int getDim()`
  `void CleanGlobalDecl()`
  `void setVerbosity(Verbosity verb)`
  `Verbosity getVerbosity()`
  `Braket Bop(std::string startid="i")`
  `Braket BopIdnum()`

  The description of these functions was done in Section 3.2.

- `bool GroupEven()`

  Returns Group parity.

- `ostream& operator<<(ostream& out, const Verbosity &a)`

  Returns current ostream verbosity level. Returns `ostream` for output.

- `size_t` **getCurrentRSS**`()`

  Returns the current resident set size (physical memory use) measured in bytes, or zero if the value cannot be determined on this OS.

- `size_t` **getPeakRSS**`()`

  Returns the peak (maximum so far) resident set size (physical memory use) measured in bytes, or zero if the value cannot be determined on this OS.

- **void print_process_mem_usage**`()`

  Prints memory stats (current and peak resident set size) in MB.

In addition, we have also implemented C++ macros that simplifies the call for the functions; its description was done in Section 3.2. Notice that the stringising macro operator `#a` causes the argument to be enclosed in double quotation marks.

- **bb**`(id)`

  This makes the call **DList**`(0, newIdx(id))`, `id` is the index in string format or enclosed in quotation marks.

- **bbt**`(id)`

  This makes the call **DList**`(1, newIdx(id))`, `id` is the index in string format or enclosed in quotation marks.

- **b**`(id)`

  This makes the call **DList**`(0, newIdx(#id))`, `id` index does not need to be enclosed in quotation marks.

- **bt**`(id)`

  This makes the call **DList**`(1, newIdx(#id))`, `id` index does not need to be enclosed in quotation marks.

- **delta** `(id1,id2)`

  This makes the call **DList**`(2, newIdx(#id1), newIdx(#id2))`; `id1` and `id2` indices do not need to be enclosed in quotation marks.

- **identity**

  This macro is a shortcut for the object **DList**`(3, 0)`.

- **bra**`(i,a,b)`

  This makes the call **Braket**`(i, #a, b, bra)`, `a` index does not need to be enclosed in quotation marks.

- **ket**`(i,a,b)`

  This makes the call **Braket**`(i, #a, b, ket)`, `i` index sum, `a` constant part without quotation marks, `b` **DList** expression.

- **braket**`(i,a,b)`

  This makes the call **Braket**`(i, #a, b, braket)`, `i` index sum, `a` constant part without quotation marks, `b` **DList** expression

- **free** `(i,a,b)`

  This makes the call **Braket**`(i, #a, b, none)`, `i` index sum, `a` constant part without quotation marks, `b` **DList** expression.

- **Field** `(a, b, c, d)`

  This makes the call **FormField**`(#a, b, c, d)`, `a` field name without quotation marks, `b` number of upper index, `c` number of lower index, `d` field with/without flavor index and symmetric/asymmetric field, returns field name in string format.

References

[1] J. C. Baez and J. Huerta, "The Algebra of Grand Unified Theories," Bull. Am. Math. Soc. **47** (2010) 483 [arXiv:0904.1556 [hep-th]].

[2] H. Georgi and S. L. Glashow, "Unity of All Elementary Particle Forces," Phys. Rev. Lett. **32** (1974) 438.

[3] H. Georgi, "The State of the Art—Gauge Theories," AIP Conf. Proc. **23** (1975) 575.

[4] H. Georgi, "Unified Gauge Theories," In *Coral Gables 1975, Proceedings, Theories and Experiments In High Energy Physics*, New York 1975, 329-339

[5] H. Fritzsch and P. Minkowski, "Unified Interactions of Leptons and Hadrons," Annals Phys. **93** (1975) 193.

[6] P. Minkowski, "$\mu \to e\gamma$ at a Rate of One Out of 10^9 Muon Decays?," Phys. Lett. B **67** (1977) 421.

[7] T. Yanagida, "Horizontal Symmetry And Masses Of Neutrinos," Conf. Proc. C **7902131** (1979) 95 [Conf. Proc. C **7902131** (1979) 95].

[8] R. N. Mohapatra and G. Senjanovic, "Neutrino Mass and Spontaneous Parity Violation," Phys. Rev. Lett. **44** (1980) 912.

[9] J. Schechter and J. W. F. Valle, "Neutrino Masses in SU(2) x U(1) Theories," Phys. Rev. D **22** (1980) 2227.

[10] M. Gell-Mann, P. Ramond and R. Slansky, "Complex Spinors and Unified Theories," Conf. Proc. C **790927** (1979) 315 [arXiv:1306.4669 [hep-th]].

[11] H. Georgi and S. L. Glashow, "Gauge theories without anomalies," Phys. Rev. D **6** (1972) 429.

[12] J. A. Harvey, P. Ramond and D. B. Reiss, Phys. Lett. B **92** (1980) 309.

[13] S. Rajpoot, "Symmetry Breaking And Intermediate Mass Scales In The So(10) Grand Unified Theory," Phys. Rev. D **22** (1980) 2244.

[14] J. A. Harvey, D. B. Reiss and P. Ramond, Nucl. Phys. B **199** (1982) 223.

[15] F. Wilczek and A. Zee, "Families from Spinors," Phys. Rev. D **25** (1982) 553.

[16] S. M. Barr, "A New Symmetry Breaking Pattern for SO(10) and Proton Decay," Phys. Lett. B **112** (1982) 219.

[17] K. S. Babu, J. C. Pati and F. Wilczek, "Fermion masses, neutrino oscillations, and proton decay in the light of Super-Kamiokande," Nucl. Phys. B **566** (2000) 33 [hep-ph/9812538].

[18] S. Bertolini, L. Di Luzio and M. Malinsky, "Intermediate mass scales in the non-supersymmetric SO(10) grand unification: A Reappraisal," Phys. Rev. D **80** (2009) 015013 [arXiv:0903.4049 [hep-ph]].

[19] M. Drees and J. M. Kim, "Neutralino Dark Matter in an SO(10) Model with Two-step Intermediate Scale Symmetry Breaking," JHEP **0812** (2008) 095 [arXiv:0810.1875 [hep-ph]].

[20] C. S. Fong, D. Meloni, A. Meroni and E. Nardi, "Leptogenesis in SO(10)," JHEP **1501** (2015) 111 [arXiv:1412.4776 [hep-ph]].

[21] R. M. Fonseca, "On the chirality of the SM and the fermion content of GUTs," Nucl. Phys. B **897** (2015) 757 [arXiv:1504.03695 [hep-ph]].

[22] K. S. Babu and S. Khan, "A Minimal Non-Supersymmetric $SO(10)$ Model: Gauge Coupling Unification, Proton Decay and Fermion masses," arXiv:1507.06712 [hep-ph].

[23] S. Rajpoot and P. Sithikong, "Implications of the SO(12) Gauge Symmetry for Grand Unification," Phys. Rev. D **23** (1981) 1649.

[24] M. Ida, Y. Kayama and T. Kitazoe, "Inclusion of Generations in SO(14)," Prog. Theor. Phys. **64** (1980) 1745.

[25] Y. Fujimoto, "So(18) Unification," Phys. Rev. D **26** (1982) 3183.

[26] D. Chang and R. N. Mohapatra, "So(18) Unification Of Fermion Generations," Phys. Lett. B **158** (1985) 323.
[27] T. Hubsch and P. B. Pal, "Economical Unification Of Three Families In So(18)," Phys. Rev. D **34** (1986) 1606.
[28] H. D. Kim and S. Raby, "Unification in 5-D SO(10)," JHEP **0301** (2003) 056 [hep-ph/0212348].
[29] F. Feruglio, K. M. Patel and D. Vicino, "Order and Anarchy hand in hand in 5D SO(10)," JHEP **1409** (2014) 095 [arXiv:1407.2913 [hep-ph]].
[30] A. Hebecker and J. March-Russell, "The structure of GUT breaking by orbifolding," Nucl. Phys. B **625** (2002) 128 [hep-ph/0107039].
[31] T. Asaka, W. Buchmuller and L. Covi, "False vacuum decay after inflation," Phys. Lett. B **510** (2001) 271 [hep-ph/0104037].
[32] T. Asaka, W. Buchmuller and L. Covi, "Gauge unification in six-dimensions," Phys. Lett. B **523** (2001) 199 [hep-ph/0108021].
[33] T. Asaka, W. Buchmuller and L. Covi, "Exceptional coset spaces and unification in six-dimensions," Phys. Lett. B **540** (2002) 295 [hep-ph/0204358].
[34] T. Asaka, W. Buchmuller and L. Covi, "Quarks and leptons between branes and bulk," Phys. Lett. B **563** (2003) 209 [hep-ph/0304142].
[35] W. Buchmuller, L. Covi, D. Emmanuel-Costa and S. Wiesenfeldt, "Flavour structure and proton decay in 6D orbifold GUTs," JHEP **0409** (2004) 004 [hep-ph/0407070].
[36] W. Buchmuller, L. Covi, D. Emmanuel-Costa and S. Wiesenfeldt, "CP Violation and Neutrino Masses and Mixings from Quark Mass Hierarchies," JHEP **0712** (2007) 030 [arXiv:0709.4650 [hep-ph]].
[37] N. Cosme and J. M. Frere, "CP violation in weak interactions from orbifold reduction: Possible unification structures," Phys. Rev. D **69** (2004) 036003 [hep-ph/0303037].
[38] Y. Hosotani and N. Yamatsu, "Gauge-Higgs Grand Unification," arXiv:1504.03817 [hep-ph].
[39] R. N. Mohapatra and B. Sakita, "SO(2n) Grand Unification in an SU(N) Basis," Phys. Rev. D **21** (1980) 1062.
[40] S. Nandi, A. Stern and E. C. G. Sudarshan, "Structure of Proton Decay in SO(n) Family Unification," Phys. Rev. D **26** (1982) 1653.
[41] P. Nath and R. M. Syed, "Analysis of couplings with large tensor representations in SO(2N) and proton decay," Phys. Lett. B **506** (2001) 68 [Phys. Lett. B **508** (2001) 216] [hep-ph/0103165].
[42] P. Nath and R. M. Syed, "Complete cubic and quartic couplings of 16 and bar-16 in SO(10) unification," Nucl. Phys. B **618** (2001) 138 [hep-th/0109116].
[43] P. Nath and R. M. Syed, "Couplings of vector-spinor representation for SO(10) model building," JHEP **0602** (2006) 022 [hep-ph/0511172].
[44] G. W. Anderson and T. Blazek, "E(6) unification model building. 3. Clebsch-Gordan coefficients in E(6) tensor products of the 27 with higher dimensional representations," hep-ph/0101349.
[45] X. G. He and S. Meljanac, "Symmetry breaking and mass spectra in supersymmetric SO(10) models," Phys. Rev. D **41** (1990) 1620.
[46] T. Fukuyama, A. Ilakovac, T. Kikuchi, S. Meljanac and N. Okada, "SO(10) group theory for the unified model building," J. Math. Phys. **46** (2005) 033505 [hep-ph/0405300].
[47] C. S. Aulakh and A. Girdhar, "SO(10) a la Pati-Salam," Int. J. Mod. Phys. A **20** (2005) 865 [hep-ph/0204097].
[48] C. S. Aulakh, "On the consistency of MSGUT spectra," Phys. Rev. D **72** (2005) 051702 [hep-ph/0501025].
[49] J. Kuipers, T. Ueda, J. A. M. Vermaseren and J. Vollinga, "FORM version 4.0," Comput. Phys. Commun. **184** (2013) 1453 [arXiv:1203.6543 [cs.SC]].
[50] P. C. West, "Supergravity, brane dynamics and string duality," In *Cambridge 1997, Duality and supersymmetric theories* 147-266 [hep-th/9811101].